Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques

Yu Cai[†] Saugata Ghose[†] Yixin Luo[‡†] Ken Mai[†] Onur Mutlu[§†] Erich F. Haratsch[‡]
[†]*Carnegie Mellon University* [‡]*Seagate Technology* [§]*ETH Zürich*

Modern NAND flash memory chips provide high density by storing two bits of data in each flash cell, called a multi-level cell *(MLC). An MLC partitions the threshold voltage range of a flash cell into four* voltage states. *When a flash cell is programmed, a high voltage is applied to the cell. Due to parasitic capacitance coupling between flash cells that are physically close to each other, flash cell programming can lead to* cell-to-cell program interference*, which introduces errors into* neighboring *flash cells. In order to reduce the impact of cell-to-cell interference on the reliability of MLC NAND flash memory, flash manufacturers adopt a* two-step programming method*, which programs the MLC in two separate steps. First, the flash memory partially programs the least significant bit of the MLC to some intermediate threshold voltage. Second, it programs the most significant bit to bring the MLC up to its full voltage state.*

In this paper, we demonstrate that two-step programming exposes new reliability and security vulnerabilities. We experimentally characterize the effects of two-step programming using contemporary 1X-nm (i.e., 15–19nm) flash memory chips. We find that a partially-programmed *flash cell (i.e., a cell where the second programming step has not yet been performed) is much more vulnerable to cell-to-cell interference and read disturb than a fully-programmed cell. We show that it is possible to exploit these vulnerabilities on solid-state drives (SSDs) to alter the partially-programmed data, causing (potentially malicious) data corruption. Building on our experimental observations, we propose several new mechanisms for MLC NAND flash memory that eliminate or mitigate data corruption in partially-programmed cells, thereby removing or reducing the extent of the vulnerabilities, and at the same time increasing flash memory lifetime by 16%.*

1. Introduction

Solid-state drives (SSDs), which consist of NAND flash memory chips, are widely used today as a primary medium of data storage. SSDs are found within large-scale data centers, consumer PCs, and mobile devices, as the per-bit cost of NAND flash memory has greatly decreased and, as a result, SSD storage capacity has greatly increased. These improvements have been enabled by both aggressive process technology scaling and the development of multi-level cell (MLC) technology. In earlier flash memory generations, each flash cell could store only a single bit of data (a single-level cell). A multi-level cell stores *two* bits of data within a single cell [3, 12, 21, 25, 37, 41], thereby doubling storage capacity.

A flash cell consists of a floating gate transistor, where the amount of charge on the floating gate determines the *threshold voltage* (V_{th}) of the cell (i.e., the voltage at which the transistor turns on). This threshold voltage falls within a range of voltage values, and the range is divided up into windows such that each window corresponds to a particular data value. For a single-level cell, the voltage range is split into *two* windows, with one representing a 0 and the other representing a 1. For a multi-level cell, the same voltage range is split into *four* windows, with each window corresponding to one of the data values 00, 01, 10, or 11. Each bit of an MLC belongs to a different flash memory *page* (the unit of data programmed and read at the same time), which we refer to, respectively, as the least significant bit (LSB) page and the most significant bit (MSB) page [5].

A flash cell is programmed by applying a large voltage on the control gate of the transistor, which triggers charge transfer into the floating gate, thereby increasing the threshold voltage. To precisely control the threshold voltage of the cell, the flash memory uses *incremental step pulse programming* (ISPP) [12, 21, 25, 41]. ISPP applies multiple short pulses of the programming voltage to the control gate, in order to increase the cell threshold voltage by some small voltage amount (V_{step}) after each step. Initial MLC designs programmed the threshold voltage in *one shot*, issuing all of the pulses back-to-back to program *both* bits of data at the same time. However, as flash memory scales down, the distance between neighboring flash cells decreases, which in turn increases the *program interference* that occurs due to cell-to-cell coupling. This program interference causes errors to be introduced into neighboring cells during programming [5, 8, 16, 26, 27, 37]. To reduce this interference by half [5], MLC NAND flash memory has been using *two-step programming* since the 40nm technology node [37].

Two-step programming stores each of the two bits within an MLC using two *separate, partial programming* steps. In its first step, two-step programming programs the LSB page: it *partially* programs the cell, setting the cell threshold voltage to a *temporary state* whose maximum voltage is approximately half of the maximum possible threshold voltage of a fully-programmed flash cell. In its second step, two-step programming programs the MSB page: it reads the LSB value into a buffer inside the flash chip (called the *internal LSB buffer*) to determine the temporary state of the cell's threshold voltage, and then partially programs the cell again, this time moving the threshold voltage from the temporary state to the desired final state. By breaking MLC programming into two separate steps, two-step programming *halves* the threshold voltage change of each MLC programming operation. Since program interference is linearly correlated with the threshold voltage change [5, 27], with two-step programming, each programming operation induces half as much interference as one-shot programming. The SSD controller interleaves the partial programming steps of a cell with the partial programming steps of neighboring cells [5, 14]. This interleaving ensures that a fully-programmed cell experiences interference only from a *single* partial programming step of a neighboring cell.

Vulnerabilities. In this paper, we find that two-step programming introduces *new possibilities* for flash memory errors that can corrupt some of the data stored within flash cells without accessing them. As there is a delay between programming the LSB and the MSB of a single cell due to the interleaved writes to neighboring cells, errors can be introduced into the already-programmed LSB page *before* the MSB page is programmed. During the second step of two-step programming, these errors are loaded *from the flash cell* into the internal LSB buffer. The LSB data in this internal buffer is used together with the data to be programmed into the MSB to determine the final voltage of the programmed cell. By buffering the LSB data inside the flash chip and not in the SSD controller, flash manufacturers avoid data transfer between the chip and the controller during the second programming step, thereby reducing the step's latency. Unfortunately, this means that the errors loaded from the internal LSB buffer *cannot* be corrected as they would otherwise be during a read operation, because the error correction engine resides only *inside the controller*, and not inside the flash chip. As a result, the final cell voltage can be *incorrectly* set during MSB programming, *permanently corrupting* the LSB data.

We study two sources of errors that can corrupt LSB data, and characterize their impact on real state-of-the-art 1X-nm (i.e., 15-19nm) MLC NAND flash chips. The first error source, *cell-to-cell program interference*, introduces errors into a flash cell when neighboring cells are programmed, as a result of parasitic capacitance coupling [5, 16, 27]. We find that program interference is a significant error source for *partially-programmed cells.* The degree of interference increases when a neighboring cell is programmed to a higher threshold voltage. As a result, we observe that the amount of interference is especially high when neighboring cells are written with *certain data patterns.* This high interference can introduce a large number of errors into the LSB page data of partially-programmed cells.

The second error source, *read disturb*, disrupts the contents of a flash cell when another cell is read [10, 15, 19, 20, 33, 42]. Several flash cells containing different pages of data are interconnected within a *flash block* (an array of flash cells that typically contains 128–256 pages), with a set of *bitlines* that connect the transistors of the flash cells in series. To accurately read the value from one transistor, the transistors belonging to the unread cells along the bitline must allow the value to *pass through.* This requires applying a voltage that is higher than the greatest possible cell threshold voltage, to guarantee that the unread cells turn on. This voltage is called the *pass-through voltage.* Unfortunately, this *pass-through voltage* is high enough to induce a *weak programming effect* on an unread cell: it slightly increases the threshold voltage of the cell [10]. As more neighboring cells within a flash block are read, an unread cell can experience enough of an increase in its threshold voltage to move its stored value to an incorrect state, as demonstrated by prior work [10, 20]. A cell is more vulnerable to read disturb if its threshold voltage is lower [10]. In two-step programming, a partially-programmed cell is more likely to have a lower threshold value than a fully-programmed cell, and thus is more vulnerable to read disturb. However, existing read disturb management solutions are designed to protect fully-programmed cells [10, 18, 39], and offer little mitigation for partially-programmed cells.

Two major issues arise from the program interference and read disturb vulnerabilities of partially-programmed and unprogrammed cells. First, the vulnerabilities induce a large number of errors on these cells, exhausting the SSD's error correction capacity and limiting the SSD lifetime (Section 4). Second, the vulnerabilities can potentially allow (malicious) applications to aggressively corrupt and change data belonging to other programs and further hurt the SSD lifetime. We present two example sketches of potential exploits that can corrupt data in this work (Section 5).

Solutions. We propose three solutions to eliminate or mitigate the program interference and read disturb vulnerabilities of partially-programmed and unprogrammed cells due to two-step programming. Our first solution eliminates the need to read the LSB page from flash memory at the beginning of the second programming step, thereby completely eliminating the vulnerabilities (Section 6.1). It maintains a copy of all partially-programmed LSB data within DRAM buffers that exist in the SSD near the controller. Doing so ensures that the LSB data is read without any errors from the DRAM buffer, where it is free from the vulnerabilities (instead of from the flash memory, where it incurs errors), in the second programming step. This solution increases the programming latency of the flash memory by 4.9% in the common case, due to the long latency of sending the LSB data from the controller to the internal LSB buffer inside flash memory. Other solutions we develop largely mitigate (but do not *fully* eliminate) the probability of two-step programming errors at much lower latency impact. Our second solution adapts the LSB read operation to account for threshold voltage changes induced by program interference and read disturb. It adaptively learns an *optimized* read reference voltage for LSB data, lowering the probability of an LSB read error (Section 6.2). Our third solution greatly reduces the errors induced during read disturb, by customizing the pass-through voltage for unprogrammed and partially-programmed flash cells. This decreases the number of errors induced by read operations to neighboring cells by 72% (Section 6.3). By eliminating or reducing the probability of introducing errors during two-step programming, our solutions completely close or greatly reduce the exposure to reliability and security vulnerabilities.

The **key contributions** of this paper are:

- We identify and experimentally characterize program errors introduced due to fundamental limitations of the two-step programming method used in MLC NAND flash memory. We analyze two major sources of these errors, program interference and read disturb, which can corrupt data stored in a partially-programmed flash cell. To our knowledge, this is the first work to characterize two-step programming and its reliability impact on real flash chips.
- We develop two sketches of new potential security exploits based on errors arising from two-step programming. Malicious applications can potentially be developed to use these exploits to corrupt data belonging to other applications.
- We propose three solutions that either eliminate or mitigate vulnerabilities to program interference and read disturb during two-step programming. One of our solutions completely eliminates the vulnerabilities, albeit with an increase in flash programming latency. Our two other solutions greatly reduce (but do not fully eliminate) the probability of data corruption, and hence minimize the vulnerabilities at a negligible latency overhead.

2. Background

In this section, we first provide necessary background on the fundamental behavior of NAND flash memory (Section 2.1). Next, we discuss how read disturb (Section 2.2) and cell-to-cell program interference (Section 2.3) induce errors in flash memory. Finally, we discuss the two-step programming algorithm used within many modern SSDs to write data into the flash memory chips within the SSD (Section 2.4).

2.1. Basics of NAND Flash Memory

A NAND flash memory chip consists of thousands of two-dimensional arrays of flash cells, known as *flash blocks.* Each block typically contains 64–128 rows (i.e., *wordlines*) of flash cells, each of which contains 64K to 128K cells. Each flash cell is a *floating gate transistor*, where stored (and programmable) charge within the floating gate determines the *threshold* *voltage* (V_{th}) at which the transistor turns on. Across different wordlines in the same block, the sources and drains of these flash cells are connected in series, forming a *bitline*.

Figure 1a shows a flash block in the commonly-used all-bit-line (ABL) flash architecture [5,12], where all cells on the same wordline are read and programmed as a single group [11] for high performance. In a *multi-level cell* (MLC) flash memory, each cell stores two bits of data. For each of the four possible two-bit values, a certain threshold voltage window is assigned, which we refer to as a *state*. Due to variation during programming, the threshold voltage of cells programmed to the same state is distributed across the state's voltage window. Figure 1b shows the expected probability density function (PDF) of multi-level cells. MLC flash memory partitions the two bits of data across two flash memory *pages* (the unit of data programmed at a time, typically 8KB). The *least significant bits* (LSBs) of all cells in a single wordline form the LSB page of that wordline (e.g., Page 1 of Wordline 1 in Figure 1a), and the *most significant bits* (MSBs) of these cells form the MSB page (e.g., Page 4 of Wordline 1 in Figure 1a). Reads and writes to the flash block are managed by the *SSD controller*.

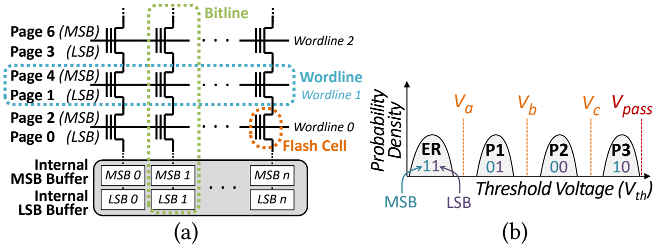


Figure 1: (a) Internal architecture of a block of all-bit-line (ABL) flash memory; (b) Threshold voltage distribution and read reference voltages for MLC NAND flash memory.

To *read* a particular page from a flash block, the controller applies a *read reference voltage* (V_{ref}) to the control gates of *all* cells on the wordline containing the page. If a cell has a V_{th} that is lower than V_{ref}, it is switched *on*; otherwise, it is switched *off*. Each bitline contains a *sense amplifier*, which can determine whether the cell is switched on or off. The value of V_{ref} depends on which page is being read from the wordline. As shown in Figure 1b, the controller applies a single V_{ref} value, V_b, to determine the LSB. If $V_{th} < V_b$, the cell is in either the ER or P1 state (as shown in Figure 1b), and holds an LSB of 1. If $V_{th} > V_b$, the cell is in either the P2 or P3 state, and holds an LSB of 0. The controller applies *two* V_{ref} values, V_a and V_c (see Figure 1b), to determine the MSB. If $V_a < V_{th} < V_c$, the cell is in either the P1 or P2 state, and holds an MSB of 0. If $V_{th} < V_a$ *or* $V_{th} > V_c$, the cell is in either the ER or P3 state, and holds an MSB of 1.

The cells on a bitline are connected in series to the sense amplifier. In order to read from a particular cell on the bitline, the controller must switch all the other *unread* cells on the same bitline *on*, to allow the value being read to propagate through to the sense amplifier. The controller achieves this by applying the *pass-through voltage* (V_{pass}) onto the wordlines of the unread cells. To ensure that unread cells always turn on, V_{pass} is set to the maximum possible threshold voltage (as shown in Figure 1b).

Before new data can be programmed, the entire block must be *erased* (due to wiring constraints). The erase operation brings the threshold voltage of each cell below 0V. Erased blocks are *opened* (i.e., selected for programming) one at a time, and a block is programmed one page at a time. The controller initially holds data to be programmed in a *DRAM buffer*, and then sends the data to buffers that reside within the flash chip (the *internal MSB and LSB buffers*; see Figure 1a). The flash chip then programs the data by repeatedly pulsing a high programming voltage ($V_{program}$) on the control gate of a cell to increase the cell's threshold voltage [5,41]. The threshold voltage increase occurs due to *Fowler-Nordheim* (F-N) *tunneling*, which allows charge to move from the substrate into the floating gate when a large differential exists between the control gate voltage and the threshold voltage [10,17]. Flash blocks can endure only a limited number of *program/erase* (P/E) *cycles*, after which data can no longer be correctly retained for the minimum amount of time guaranteed [6,30]. At this point, we say that the flash block is *worn out*.

2.2. Read Disturb

Read disturb is the phenomenon where a read to one data page within the flash block can alter the threshold voltages of the other *unread* data pages in the same block [10,15,19,20,33, 42]. When V_{pass} is applied to the cells of an unread page (see Section 2.1), it induces a *weak programming effect* on these cells. While V_{pass} is lower than $V_{program}$, it is still high enough to induce a small amount of F-N tunneling. This tunneling shifts the threshold voltage of the unread cells higher, and has a larger impact on cells with lower threshold voltages [10].

2.3. Cell-to-Cell Program Interference

While programming pulses (see Section 2.1) are applied to one wordline within the block, they can induce errors on cells that belong to *adjacent* wordlines of the same block [5,8,16,26, 27,37]. This phenomenon is referred to as *cell-to-cell program interference*, and occurs due to *parasitic capacitance coupling* within the block [5,27]. As a result of the coupling, as the threshold voltage of a cell increases during programming, the threshold voltages of cells on adjacent wordlines (which we call *victim cells*) increase as well. Eventually, the unintended voltage increases can move a victim cell into a different state than the one it was originally programmed in, resulting in an error when the data is subsequently read from the cell. Cell-to-cell program interference is one of the most critical scaling barriers for NAND flash memory [3,24,33,37].

2.4. Two-Step Programming

To reduce the impact of cell-to-cell program interference, two-step programming has been adopted for sub-40 nm flash memories [5,37]. Two-step programming writes the values of the two pages within a wordline in two independent stages. Figure 2 shows how the cell state and cell threshold voltage level change during two-step programming. After an erase operation on a block, all flash cells are at the erased (ER) state. In the first step of two-step programming, *only* the LSB page is programmed. If the LSB of a cell should be 0, the flash cell is programmed into a temporary program state (TP); otherwise, it remains in the ER state. In the second step, the MSB page is programmed. For an MSB of 1, the cell either remains in the ER state (if its LSB is 1), or moves up from the TP state to the P3 state (if its LSB is 0). For an MSB of 0, the cell either moves up from the ER state to the P1 state (if its LSB is 1), or moves up from the TP state to the P2 state (if its LSB is 0).

Recall from Section 2.3 that the amount of program interference increases with the threshold voltage change during programming. To reduce the interference induced on neighboring wordlines, the LSB and MSB pages are *not* programmed back-to-back. Instead, two-step programming uses *shadow*

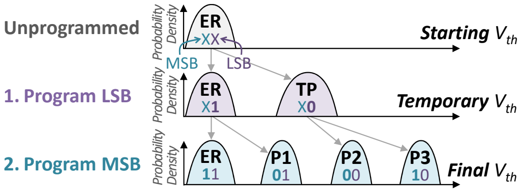


Figure 2: Starting (after erase), temporary (after LSB programming), and final (after MSB programming) states for two-step programming.

program sequencing [5, 14] to interleave the programming steps of *multiple wordlines* with each other. For an arbitrary Wordline i (e.g., Wordline 1 in Figure 1a), shadow program sequencing ensures that the wordline is fully programmed (i.e., the second programming step is performed) only after both pages of Wordline $i-1$ (e.g., Pages 0 and 2 of Wordline 0 in Figure 1a) and the LSB page of Wordline $i+1$ (e.g., Page 3 of Wordline 2) are programmed.

In shadow program sequencing, each page within a block is assigned a unique page number. The LSB page of Wordline i is numbered Page $2i-1$ (e.g., Page 1 in Wordline 1 in Figure 1a), and the MSB page is numbered Page $2i+2$ (e.g., Page 4 in Wordline 1). The only exceptions are the LSB page of Wordline 0 (Page 0) and the MSB page of the last wordline n (Page $2n+1$). Two-step programming interleaves the programming steps by writing to the pages in increasing order of page number [5, 14]. Thus, once a wordline is fully programmed (e.g., Page 4 in Wordline 1 is programmed), it experiences interference from only one single programming step of a neighboring wordline (e.g., Page 6 in Wordline 2).

3. Motivation

In this section, we first discuss how the error rate of the data stored in NAND flash memory can affect the lifetime of an SSD (Section 3.1). We then discuss how two-step programming can introduce errors into partially-programmed data (Section 3.2), thereby potentially limiting the SSD lifetime.

3.1. Raw Bit Error Rate, ECC, and Lifetime

Within NAND flash memory, the number of *raw bit errors* (i.e., the number of errors before data is corrected) limits the total *lifetime* of the memory. Raw bit errors are corrected using *error-correcting code* (ECC) mechanisms within the SSD controller. As the ECC engines are typically in hardware, they have a maximum *error correction capability* (i.e., they can *correct only* up to a fixed number of errors, N, for every read). As more P/E cycles are performed on the NAND flash memory, its raw bit error rate increases [3, 4, 9, 10]. As a result, the number of errors eventually exceeds N at a certain P/E cycle count. This P/E cycle count is defined to be the SSD *lifetime*, which we show in Figure 3 as *Normal Lifetime*. If more errors are introduced into the memory, for example, by maliciously taking advantage of vulnerabilities that arise from two-step programming, the lifetime of the SSD can be shortened, to a P/E cycle count shown as *Reduced Lifetime* in Figure 3. This is because the error correction capability N remains fixed, yet now there are *more* raw bit errors to correct. In the rest of this paper, we examine the impact of the errors that can be introduced by two-step programming (Section 4), and how malicious software can potentially induce and exploit these errors to shorten the lifetime of an SSD (Section 5).

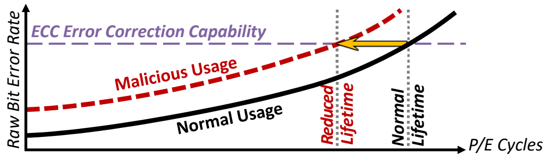


Figure 3: Flash lifetime reduction due to inducing more errors, via potentially malicious usage.

3.2. Error Sources in Two-Step Programming

Between the first and second steps of two-step programming, a partially-programmed wordline holds only LSB data. During this time, cell-to-cell program interference and read disturb can introduce raw bit errors into the LSB data. These errors can cause the wordline to be programmed to an incorrect state in the second programming step.

During the second step, both the MSB and LSB of each cell in the wordline are required to determine the final target state of the cells. As shown in Figure 4, the data to be programmed into the MSB is loaded from the SSD controller to the internal MSB buffer (① in the figure). Concurrently, the LSB data is loaded into the internal LSB buffer from the flash memory wordline (②). The LSB data never goes to the controller before being read into the internal LSB buffer, as Figure 4 shows, and is thus *unable* to use the ECC engine within the controller (③). As a result, the raw bit errors in the LSB data cannot be corrected, and translate into *program errors*.

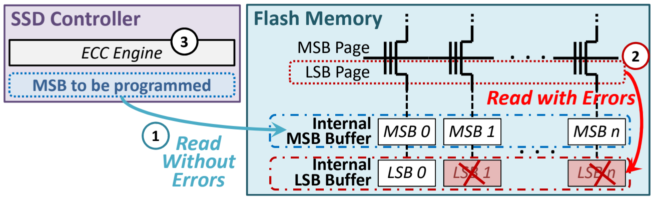


Figure 4: In the second step of two-step programming, LSB data does not go to the controller, and is not corrected when read into the internal LSB buffer, resulting in program errors.

We classify all possible program errors into four categories based on their LSB and MSB values, as shown in Table 1. For example, let us assume that a cell is supposed to be set to the P1 state, with an MSB of 0 and an LSB of 1 (ERR2 in Table 1). However, the LSB data in the flash chip contains errors, and is incorrectly read as a 0. As a result, the flash chip proceeds to believe that both the MSB and the LSB of the cell should be 0, and *incorrectly* increases the cell threshold voltage to the P2 state.

Table 1: Program error types caused by LSB read errors.

Category	Correct Cell State	MSB	LSB		Programmed Cell State
			Original	*Misread As*	
ERR1	ER (11)	1	1	0	P3 (10)
ERR2	P1 (01)	0	1	0	P2 (00)
ERR3	P2 (00)	0	0	1	P1 (01)
ERR4	P3 (10)	1	0	1	ER (11)

Recall from Section 2.4 that when a cell is partially programmed with an LSB of 0, its threshold voltage increases to the TP state (see Figure 2). As Table 1 shows, such a cell can experience either category ERR3 or ERR4 errors. Programming can only *increase* the voltage of a partially-programmed cell. For ERR3 and ERR4, the SSD controller *attempts* to set the cell to states P1 and ER, respectively. However, since the cell is already in the TP state, its voltage remains unchanged. Furthermore, it is unlikely for a cell in the TP state to be misread as a cell in the ER state, as both cell-to-cell program interference and read disturb shift the cell threshold voltage to a *higher* value, not a lower one. As a result, ERR3 and ERR4 errors are much less likely to occur than ERR1 and ERR2 errors, as shown in prior work [3]. Therefore, we primarily focus on ERR1 and ERR2 in this work.

Program errors introduced by two-step programming can lead to data corruption within the LSB page. In this paper, our goal is to demonstrate that these program errors significantly reduce the lifetime of an SSD. To this end, we use real flash memory chips to quantify the extent to which program interference and read disturb lead to such errors in Section 4.

4. Characterization of Error Sources

In this section, we study the two major sources of errors that are introduced into partially-programmed wordlines during two-step programming, by characterizing the impact of each error source using real state-of-the-art NAND flash memory chips. We first briefly discuss our characterization methodology in Section 4.1. We next characterize the impact of cell-to-cell program interference in Section 4.2. We then characterize the impact of read disturb in Section 4.3.

4.1. Methodology

We characterize multiple state-of-the-art 1X-nm (i.e., 15–19nm) flash chips using an FPGA-based flash testing platform [2] that allows us to issue commands directly to raw chips. In order to determine the threshold voltage stored within each cell, we use the *read-retry* mechanism built into modern SSD controllers [5, 9, 40, 47]. Throughout this paper, we present normalized voltage values, as actual voltage values are proprietary information to flash vendors.

4.2. Cell-to-Cell Program Interference

Recall from Section 2.4 that cell-to-cell program interference causes the threshold voltages of victim cells to increase when an adjacent cell is being programmed. Two-step programming using shadow program sequencing can halve the program interference to the MSB page [37], as an MSB page of Wordline n experiences interference only from a single neighbor (partially programming the MSB page of Wordline $n + 1$). The LSB page of Wordline n, however, experiences interference from *two* neighbors (partially programming the MSB page of Wordline $n - 1$ and the LSB page of Wordline $n + 1$) while Wordline n remains partially programmed.

Due to continued scaling, cell-to-cell interference on the LSB page is no longer negligible. For example, after the LSB page on Wordline 1 (Page 1 in Figure 1a) is programmed, the next two pages that are programmed (Pages 2 and 3) reside on directly-adjacent wordlines. Therefore, before the MSB page on Wordline 1 (Page 4 in Figure 1a) is programmed, the LSB page could be *susceptible* to program interference, and its threshold voltage distribution widens, as shown in Figure 5. The widened distribution can cause some of the cells to be misread. For example, after the distribution of the ER state shifts in Figure 5, some of the cells belonging to the state (e.g., the red dot in the figure) now fall on the other side of the read reference voltage (V_{ref}), and are incorrectly read as being in the TP state. Such read errors eventually translate to program errors during the second step, MSB page programming (see Section 3.2). Our goal in this section is to quantify the impact of program interference on the LSB page of a partially-programmed wordline.

First, we characterize the threshold voltage change of an LSB page due to program interference after programming Page 1 (see Figure 1a) with pseudo-random data. We use pseudo-random data to mimic the data scrambling employed by modern SSDs [13, 22]. Figure 6a shows the measured distribution of cells in the ER state after four different times:

A. Just after Page 1 is programmed (no interference),
B. Page 2 is programmed with pseudo-random data,
C. Pages 2 and 3 are programmed with pseudo-random data,
D. Pages 2 and 3 are programmed with a data pattern that induces the *worst-case* program interference (see below).

We study only the ER state, as ERR1 and ERR2 errors occur only when the LSB is 1 (see Section 3.2). The programming of Pages 2 and 3 shifts the threshold voltage distribution to the right, and widens the distribution. Ideally, the threshold voltage of all cells in the ER state should be below 0V, but as we observe, program interference increases the number of cells whose threshold voltage is greater than 0V.

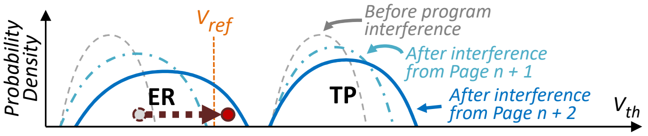
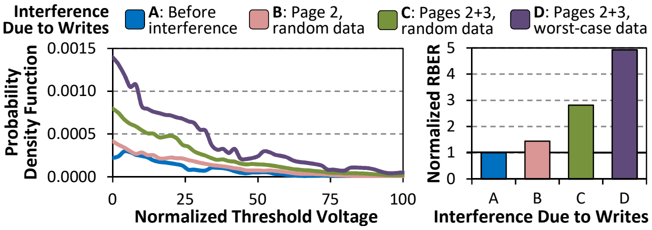

Probability Density
V_{ref}
Before program interference
After interference from Page n + 1
After interference from Page n + 2
ER
TP
V_{th}

Figure 5: Threshold voltage distribution of partially-programmed cells in Page *n* after different amounts of program interference by neighboring wordlines. Red dot shows a cell that is misread after program interference.



(a) (b)

Figure 6: ER state threshold voltage distribution (a) and normalized raw bit error rate (b) of partially-programmed Page 1 within a flash block with 8K P/E cycles, before and after cell-to-cell program interference.

Second, we measure the number of errors in an LSB page when the second step of two-step programming (MSB page programming) begins. When MSB programming takes place (e.g., Page 4 in Figure 1a), the flash memory relies on the default read reference voltage (e.g., 0V) to read the LSB page (Page 1) data into the internal LSB buffer (see Section 2.4). Due to flash interface limitations, we *cannot* directly measure the read errors on the LSB page when the MSB page is being programmed, as the internal LSB buffer cannot be accessed from outside the flash memory. As a workaround, before programming Page 4, we directly read the partially-programmed Page 1 out to the controller using the default read reference voltage, and measure the raw bit error rate (RBER). Figure 6b shows the RBER of partially-programmed Page 1, normalized to the RBER without any cell-to-cell program interference. We see that after Pages 2 and 3 are programmed with pseudo-random data (C in Figure 6b), the RBER is 2.8x the interference-free RBER. This means that when Page 4 is programmed, the data in the internal LSB buffer contains 2.8 times the number of errors due to program interference. Therefore, program interference on a partially-programmed wordline greatly increases the probability that a program error occurs during MSB page programming.

Third, we explore how the error rate changes if, instead of writing *random* data to Pages 2 and 3, we use the *worst-case* data pattern. We construct the worst-case pattern using two operations. First, we program the inverse of Page 0 into Page 2 (an MSB page), to induce the largest possible threshold voltage change on Wordline 0 (from the ER state to the P1 state, or from the TP state to the P3 state). Second, we program Page 3 to all 0s, which sets the page to the TP state. This induces the largest possible voltage differential to raise the Page 1 threshold voltages. Note that *the worst-case pattern requires no knowledge of the data stored within Page 1.* Compared to pseudo-random data (C), the worst-case pattern (D) causes an even larger threshold voltage increase for the ER state (Figure 6a), and thus leads to a higher RBER (Figure 6b). The RBER with the worst-case pattern is 4.9x that without interference (Figure 6b). The RBER is significantly higher because the larger voltage changes of the worst-case pattern induce a greater degree of program interference.

Summary. We conclude that cell-to-cell program interference can significantly increase the number of LSB errors introduced during two-step programming, thus hurting reliability once the wordline is fully programmed.

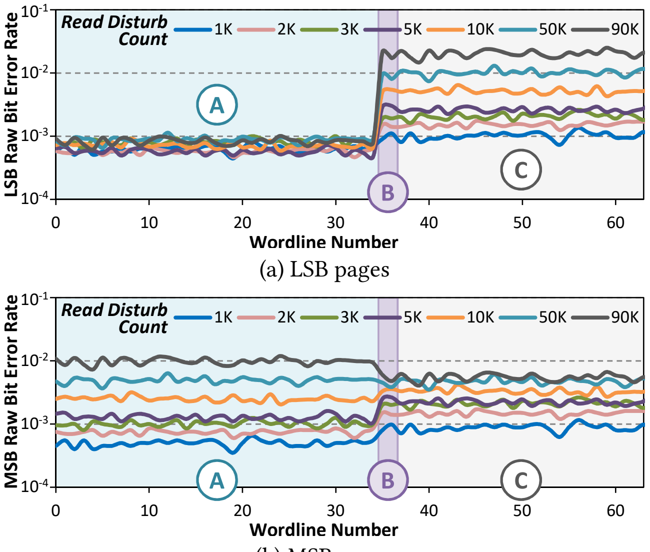
4.3. Read Disturb

Recall from Section 2.2 that read disturb causes the threshold voltages of unread cells to increase when a cell in the same block is being read. Read disturb results in a larger threshold voltage shift for a cell with a lower initial threshold voltage [10]. Thus, *unprogrammed and partially-programmed cells are more likely to experience errors from read disturb*, as they have lower threshold voltages. Once a block is opened for writing, it takes time to write to all of the hundreds of pages within the block. During this time, an open block includes some fully-programmed cells, some partially-programmed cells, and some unprogrammed cells. Thousands of reads can access the open block, each of which induces read disturb on pages other than the one that is being read.

Our goal is to quantify the impact of read disturb on unprogrammed and partially-programmed cells in an open block. To replicate an open block in our experiments, we program the first 72 pages (Pages 0–71) of a 256-page block, which results in 35 wordlines (Wordlines 0–34) with fully-programmed cells, two wordlines with only their respective LSB pages programmed (Wordlines 35 and 36), and all other wordlines unprogrammed. We repeatedly read one of the fully-programmed pages (Page 0) N times. N corresponds to the *read disturb count* experienced by each page (except Page 0) in the block. Then, we program all of the remaining pages within the block, and observe how the bit error rate differs for pages that were unprogrammed or partially programmed during the repeated reads, compared to bit error rate for fully-programmed pages. We repeat the experiments by varying N (between 1,000 and 90,000 repeated reads).

Figure 7a shows the RBER for each LSB page in the first 64 wordlines of a block with 8K P/E cycles, for various read disturb counts on Page 0. Figure 7b shows the RBER under the same situation for each MSB page. The figures show representative behavior for an open block that has been exposed to read disturb. We sort the pages by their corresponding wordline number on the x-axis, and each line represents a different read disturb count. We make three observations from these figures.

First, *read disturb induces errors not only in wordlines that were programmed before the read disturb took place, but also in wordlines that are not yet programmed.* During read disturb, the threshold voltage of a cell in an unprogrammed wordline (which is initially below 0V) increases. With a high enough read disturb count, the cell can move from the ER state to the P1 state, and hence cannot correctly store the value 11 when it is programmed. As a result, LSB pages in unprogrammed wordlines (whose error rates are represented by the lines that fall into Region Ⓒ in Figures 7a and 7b) exhibit much higher error rates than LSB pages in fully-programmed wordlines (whose error rates are represented by the lines that fall into Region Ⓐ). Some MSB pages, especially the ones subject to lower read disturb counts, exhibit this trend as well.



(a) LSB pages

(b) MSB pages

Figure 7: Raw bit error rate for the pages in the first 64 wordlines of a block, for various read disturb counts. At the time that the read disturb occurs, Wordlines 0–34 (Region Ⓐ) are fully programmed, Wordlines 35–36 (Region Ⓑ) are partially programmed, and Wordlines 37–63 (Region Ⓒ) are unprogrammed.

Second, *LSB pages in partially-programmed and unprogrammed wordlines experience an order of magnitude increase in raw bit error rate when compared to LSB pages in fully-programmed wordlines.* This can be observed by comparing the RBER for Regions Ⓑ and Ⓒ to the RBER of Region Ⓐ in Figure 7a. As can be seen in Figure 2, a cell in a partially-programmed or unprogrammed wordline has, on average, a much lower threshold voltage than a cell in a fully-programmed wordline. This is because a partially-programmed cell can only be in either the ER state (whose threshold voltage is negative, and the farthest away from V_{pass}) or the TP state (whose threshold voltage is approximately half that of V_{pass}), and an unprogrammed cell starts off in the ER state. In contrast, a fully-programmed cell can have a threshold voltage almost as high as V_{pass}, which greatly reduces the threshold voltage change induced by read disturb. Hence, LSB pages in partially-programmed and unprogrammed wordlines are much more susceptible to read disturb than LSB pages in fully-programmed wordlines.

Third, *LSB pages in partially-programmed and unprogrammed wordlines experience approximately 2x the error rate of MSB pages in fully-programmed wordlines.* Previous studies [10], which characterized errors on only fully-programmed wordlines, found that MSB pages (Region Ⓐ in Figure 7b) were the most sensitive pages to read disturb. In our characterization, we find that the LSB pages in partially-programmed and unprogrammed wordlines (Regions Ⓑ and Ⓒ in Figure 7a, respectively) are the most sensitive pages, experiencing twice as many errors at each read disturb count. Thus, the higher RBER of the LSB pages of partially-programmed and unprogrammed wordlines is the limiting factor for read disturb tolerance.

Summary. We conclude that in 1X-nm flash chips, the LSB pages in partially-programmed and unprogrammed wordlines are significantly more vulnerable to read disturb errors than other types of pages. By temporarily creating partially-programmed wordlines, two-step programming greatly increases the impact of read disturb on an open flash block, which negatively affects the reliability of the flash memory.

5. Security Exploit Sketches

When multiple applications share an SSD, the data from the different applications is stored within the same physical flash block if they issue writes one after another. This is because the SSD typically maintains only one open block per die to store data from the host machine, and uses shadow program sequencing (see Section 2.4). All pages of an open block are fully programmed before a new block is opened. As discussed in Section 3.2 and demonstrated on real flash chips in Section 4, a page in a flash block can induce errors into another page in the same block, especially to the LSB pages in partially-programmed or unprogrammed wordlines. As a result, an application can potentially circumvent file protection permissions by exploiting low-level interference effects during the underlying two-step programming. In this way, a malicious application can inject errors into the files of other users, potentially leading to data corruption.

In the rest of this section, we show two high-level examples of system exploits that take advantage of the errors introduced by two-step programming, based on the error sources we demonstrated in Section 4.[1] We provide more detail on our exploits, especially on how we overcome difficulties in their implementations, in the extended version of this work [1].

5.1. Sketch of Program Interference Based Exploit

In this sketch of a new exploit, we describe how a malicious application can induce a significant amount of *program interference* onto a flash page that belongs to another, benign victim application, corrupting the page and shortening the SSD lifetime. Recall from Section 4.2 that due to program interference, writing the worst-case data pattern (i.e., all 0s) can induce 4.9x the number of errors into a neighboring page (with respect to an interference-free page) without needing to know the contents of the neighboring page that is disrupted. The goal of this exploit is for a malicious application to write this worst-case pattern in a way that ensures that the page that is disrupted belongs to the victim application, and that the page that is disrupted experiences the greatest amount of program interference possible.

Exploit Walkthrough. We now discuss an example of the exploit, where the page under attack (i.e., the page belonging to the victim application) is written to the LSB page of an arbitrary Wordline n within an open flash block. Figure 8 illustrates the contents of the pages within Wordline n and its surrounding wordlines during the example exploit. The malicious application first prepares for the exploit by writing a small 16KB file,[2] consisting of all 1s, to the SSD. The SSD controller writes the file to the open block in shadow program sequence order (see Section 2.4), in Pages $2n-3$ and $2n-2$ (① in Figure 8). Importantly, this keeps the cells of partially-programmed Wordline $n-1$ in the ER state. The malicious application then waits for the victim application to write data to the page under attack (Page $2n-1$; ②), by monitoring the file system to check when the victim application's file size increases (this is unprotected information).

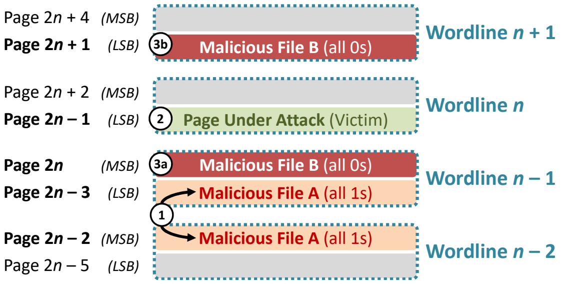
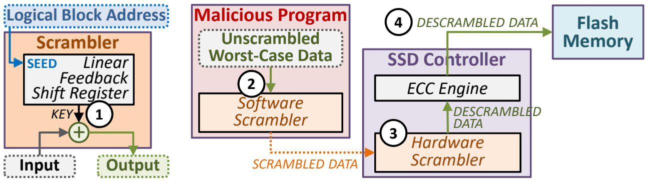


Figure 8: Layout of data within a flash block during a program interference based exploit.

Once the page under attack is written, Wordline n is partially programmed. The malicious application then induces the maximum possible program interference onto Wordline n by writing a second 16KB file, containing all 0s, to the SSD. The SSD controller writes the file to Pages $2n$ and $2n+1$ (③a and ③b). Recall from Section 4.2 that the amount of interference induced on a victim page is greatest when a neighboring wordline has the largest possible threshold voltage change. Writing all 0s to Page $2n$ causes each flash cell in Wordline $n-1$ to go from the ER state to the P1 state, which is the largest threshold voltage change possible when writing to an MSB page. Writing all 0s to Page $2n+1$ causes each flash cell in Wordline $n+1$ to go from the ER state to the TP state, which is the largest threshold voltage change possible when writing to an LSB page. Thus, the malicious application maximizes the amount of program interference induced on Wordline n, increasing the probability of bit flips in the page under attack.

Scrambling Workaround. To exert the maximum possible interference, the malicious application needs to write specific data values to the SSD. One obstacle is that many SSDs internally *scramble* data before storing the data in the flash memory [13, 22]. Scrambling randomizes the number of 0s and 1s, to reduce the effects of data pattern dependence on flash errors. Figure 9a shows the design of the scrambler. The scrambler first uses a *linear feedback shift register* (LFSR) to generate a scrambling key using an input seed (usually the logical address of the data) [13, 28]. It then XORs this key with the data (① in Figure 9a). Data is descrambled using the same hardware, with the same key, as the XOR is reversible.



(a) Scrambler (b) Workaround of in-controller scrambling

Figure 9: Malicious applications can use a software scrambler to reverse the in-controller scrambling process.

The malicious application can take advantage of the reversible behavior to circumvent scrambling, as shown in Figure 9b. The application recreates the scrambler in software (② in

[1] Due to limited space, we do not discuss *superblocks* (i.e., multiple blocks striped across the multiple dies within a chip) or *superpages* (i.e., each stripe of flash pages within a superblock). Our exploits can be extended easily to work in the presence of superblocks, by treating the superblock and superpage as a larger block and page, respectively.

[2] We assume, without loss of generality, that each flash page holds 8KB of data. Thus, our 16KB file spans two flash pages.

Figure 9b). It can generate the key easily, as the logical address of its file can be found (e.g., using the Linux commands `hdparm -fibmap` or `debugfs -R`), and the base polynomial used by the LFSR can be determined [43]. This allows the application to generate the scrambled version of the data, which it then writes to the SSD (③). The hardware scrambler in the SSD controller XORs the data with the same key, which descrambles the data back to its original value (④). The descrambled data is then stored in the flash cells.

We describe the detailed operation of modern data scramblers and our scrambling workaround in the extended version of this work [1].

5.2. Sketch of Read Disturb Based Exploit

In this sketch of a new exploit, we describe how a malicious application can induce a significant amount of *read disturb* onto *several* flash pages that belong to other, benign victim applications. As discussed in Section 4.3, two-step programming greatly increases the vulnerability of both partially-programmed and unprogrammed flash cells to read disturb, with an error rate double that of the worst-case pages in fully-programmed cells. The goal of this exploit is for a malicious application to quickly perform a large number of read operations in a very short amount of time, to induce read disturb errors that corrupt both pages already written to partially-programmed wordlines and pages that *have yet to be written.*

Exploit Walkthrough. We now walk through an example of the exploit. The malicious application writes an 8KB file, with arbitrary data, to the SSD. The SSD controller writes the file to a page within an open flash block (e.g., Page 4 in Figure 1a). Immediately after the file is written, the malicious application repeatedly issues the system calls *fopen()*, *fread()*, *fflush()*, and *fclose()* for the file it has just written, forcing the file system to send a new read request to the SSD every time the sequence of system calls is invoked. The malicious application continuously performs the system call sequence for a short period of time (e.g., a few seconds), and then stops.

As each system call sequence is issued, read disturb is induced on the other wordlines within the flash block, causing the cell threshold voltages of these wordlines to increase. In particular, the threshold voltages of cells in the unprogrammed wordlines can increase to a point where the cells can no longer be programmed to the ER state (see Section 4.3). After the malicious application finishes performing the repeated system call sequence, a victim application writes data to a file. The SSD writes this data to the open flash block that was just disturbed by the malicious application. As the SSD is unaware that an attack took place, it does not detect that the data cannot be written correctly. As a result, bit flips can occur in the victim application's data.

In this attack, the malicious application does *not* need to know any information about the victim application. The malicious application can inject errors into locations where data will be stored by other applications *in the future.* Unlike the program interference exploit, which attacks a single page, the read disturb exploit can corrupt multiple pages with a single attack, and the corruption can affect pages written at a much later time than the attack if the host write rate is low.

Performing Rapid Reads on the SSD. The malicious application can perform a large number of system call sequences in a short amount of time. Flash memory can execute a one-page read operation in 100 µs. Therefore, the application can issue approximately 10K read operations to the SSD every second. The caching of frequently-accessed pages in the SSD, as done in some modern SSDs [23,32], can potentially prevent an application from inducing many reads to the flash chip. This cache is small in modern SSDs, to minimize cost, and the malicious application can easily render the cache ineffective by issuing a small number of reads to other pages during each system call sequence [1]. Thus, each sequence can still induce read disturb on the flash chip.

Existing Refresh and Read Disturb Management. Refresh mechanisms can eliminate read disturb errors by periodically correcting and rewriting data stored within the SSD [6,34,36]. The burst of system call sequences can happen so fast that existing refresh techniques cannot avoid them, as refresh is triggered much less frequently (e.g., weekly or monthly [6,29,30]). Existing read disturb management techniques (e.g., [18,39]) are also ineffective for such an exploit, as they mitigate read disturb errors on a block only when the error rate of a *fully-programmed* page is too high, and are unaware of the error rate on an unprogrammed page.

6. Protection and Mitigation Mechanisms

As we have seen, errors introduced by two-step programming can shorten the SSD lifetime, and have the potential to expose security exploits. If the error sources can be controlled, the reliability and (potential) security vulnerabilities can be removed or greatly reduced. To this end, we propose three solutions that either eliminate or restrict the impact of cell-to-cell program interference and read disturb during two-step programming. Table 2 summarizes the cost and benefits of each mechanism. We provide more detail on our mechanisms in the extended version of this work [1].

Table 2: Summary of our proposed protection mechanisms.

Mechanism	Protects Against	Overhead	Error Rate Reduction
Buffering LSB Data in the Controller (§6.1)	interference read disturb	2MB storage 1.3–15.7% latency	100%
Adaptive LSB Read Reference Voltage (§6.2)	interference read disturb	64B storage 0.0% latency	21–33%
Multiple Pass-Through Voltages (§6.3)	read disturb	0B storage 0.0% latency	72%

6.1. Buffering LSB Data in the Controller

The goal of our first mechanism is to eliminate *all* of the errors that can corrupt an LSB page during the second step of two-step programming. Errors exist during two-step programming because the flash chip must read the LSB page data *directly from the flash cells* into an internal LSB buffer *before* programming the MSB page data. The LSB page is highly vulnerable, as we demonstrated in Section 4, and can contain many errors. As this data does *not* go through the SSD controller before the second programming step, these errors *cannot* be corrected using the built-in ECC that sits in the controller. Thus, incorrect LSB data is used to decide the final state of the cell during the second programming step. If the second step can be modified to avoid *directly reading LSB data from the flash cells*, we can eliminate these errors. The key idea of our solution ensures that the LSB page data is *not* read from the flash chip, but is buffered in and read from some dedicated location in the SSD.

Mechanism. We propose to use the DRAM buffer in the SSD[3] to store the LSB data for all partially-programmed wordlines. We modify two-step programming to make use of the LSB data buffered in DRAM. The key idea is to ensure that LSB data from partially-programmed wordlines is read from

[3] Many SSDs already have additional DRAM to manage logical-to-physical address mapping and other metadata [32].

this buffer and *not* from the flash chip during the second programming step, as the DRAM buffer does not experience the LSB read errors that we aim to eliminate.

Figure 10 shows our modified two-step programming algorithm. For the first programming step (LSB page), the algorithm is largely unchanged. In the original two-step programming algorithm, the LSB data was released from the DRAM buffer once the data was sent to the flash chip (Step A in Figure 10). We modify the algorithm to keep the data within the DRAM buffer (Step B). During the second programming step (MSB page), the controller now looks up the DRAM buffer to see if the buffer has the LSB data (Step C). If the LSB data is found, the controller retrieves the data from DRAM (Step D) and sends it to the flash chip's internal LSB buffer (Step E). Then, the controller sends the MSB data to be programmed to the flash chip's internal MSB buffer (Step F). This eliminates the need to read data *directly* from the flash chip, thus eliminating the risk of reading data that contains errors due to program interference or read disturb. The flash memory then programs the cells in the wordline.

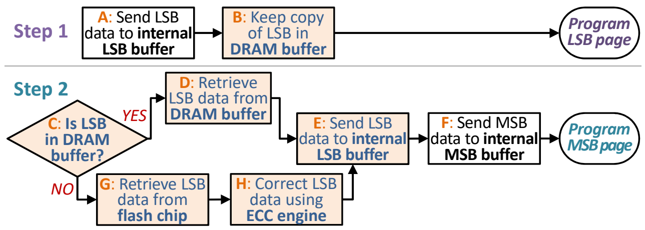

Step 1
A: Send LSB data to internal LSB buffer
B: Keep copy of LSB in DRAM buffer
Program LSB page
Step 2
C: Is LSB in DRAM buffer?
YES
D: Retrieve LSB data from DRAM buffer
E: Send LSB data to internal LSB buffer
F: Send MSB data to internal MSB buffer
Program MSB page
NO
G: Retrieve LSB data from flash chip
H: Correct LSB data using ECC engine

Figure 10: Modified two-step programming, using a DRAM buffer for LSB data (modifications shown in shaded boxes).

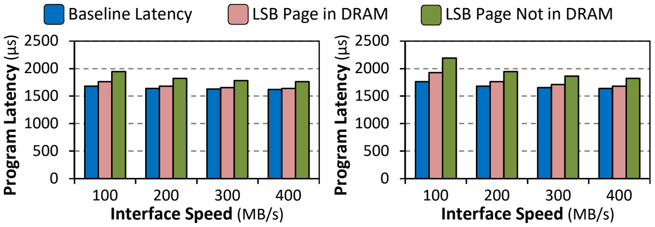
There are some uncommon cases where the DRAM buffer does not contain the LSB data. For example, if the SSD is turned off and then turned on again in between the two programming steps, the DRAM buffer is cleared. In this situation, the algorithm reads the LSB data from the flash chip (Step G in Figure 10), and then dispatches it to the SSD controller, where ECC corrects the bit errors in the data (Step H). Once the controller gets the error-free LSB data, it sends the LSB and MSB data to the flash chip for programming (Steps E and F). While reading from the flash chip and applying ECC takes too long to perform regularly, it can be used as a fail-safe mechanism for infrequent events such as power cycling.

Errors Removed. Our mechanism eliminates the errors in the LSB page before the second step starts, either by (1) reading data from the DRAM buffer, where it is free from the the error sources we discussed in Section 3.2; or (2) reading the LSB page into the SSD controller and *correcting* its errors. As a result, all errors introduced due to the two-step programming method are removed, thus eliminating any associated reliability and security vulnerabilities.

Storage Overhead. For every block, no more than two LSB pages need to be stored in the DRAM buffer at any given time, as shadow sequencing ensures that there are at most two partially-programmed wordlines. Within each die of a flash chip, only one block is open for writing at a time. Therefore, we require two pages' worth of storage per die in the DRAM buffer. For a 1TB SSD with 64 dies, with a 16KB page size, the total DRAM storage required is 2MB. A modern 1TB SSD typically comes with at least 1GB of DRAM. Our proposal occupies only 0.2% of this existing DRAM storage.

Latency Overhead. Our mechanism requires additional time to load LSB data into the internal LSB buffer during the second step of programming. A DRAM lookup must be performed every time an MSB page is programmed. If the LSB data is *not* in DRAM, we incur extra latency to read and correct the data. In both cases, the LSB data must be transmitted from the SSD controller to the internal LSB buffer. The dominant latency components of our mechanism are MSB page programming time (1.6 ms), transferring data between the controller and the flash memory (for a 100MB/s flash interface, 160 µs per 16KB page), and the page read operation when the DRAM buffer does *not* have the LSB data (100 µs).

Figure 11 shows the latency overhead of performing MSB page programming, for baseline two-step programming and our DRAM-buffer-based programming mechanism (showing both when the data is in DRAM, and when it is not and must be read from the flash cell). For a 100MB/s flash interface, we see that the latency overhead is 4.9% when the LSB page is in the DRAM buffer, and 15.7% when the page is not in the DRAM buffer, assuming a commonly-used 8KB page size (Figure 11a). For a more conservative 16KB page, these latency overheads are 9.3% and 24.2%, respectively (Figure 11b). We also evaluate various flash interface speeds, as recent flash standards propose speeds as high as 400MB/s [35]. We see that the overhead drops significantly, to 1.3% and 8.7%, respectively, for the more common 8KB page size (Figure 11a), and to 2.5% and 11.1% for a 16KB page size (Figure 11b).



(a) 8KB page size

(b) 16KB page size

Figure 11: MSB programming latency when LSB data is buffered in the SSD controller, for different page sizes and interface speeds.

We conclude that maintaining a buffer of LSB pages within the SSD controller is an effective option for eliminating program errors, and that the latency overhead can be reasonably small, depending on flash page size and interface speed.

6.2. Adaptive LSB Read Reference Voltage

The goal of our second mechanism is to reduce errors as much as possible at minimal latency overhead. To this end, we develop a mechanism that significantly reduces read errors that occur for a partially-programmed page. As discussed in Section 3.2, errors are introduced into the LSB page data when the threshold voltage of a partially-programmed flash cell shifts upwards. Traditionally, during MSB page programming, a *fixed* read reference voltage is applied to read the LSB page, under the assumption that the threshold voltage distribution does not change significantly. However, the threshold voltage distribution shifts not only due to the errors introduced due to two-step programming, but also due to wear-out, leading to read errors when a fixed voltage is used for the LSB page. The key idea of our mechanism is to optimize the read reference voltage that is used for LSB pages, such that the new read reference voltage accounts for threshold voltage shifts and thus reduces the number of raw bit errors when the page is read from a partially-programmed wordline.

Mechanism. We propose to use an *adaptive* read reference voltage for the partially-programmed LSB pages. Once per day, this adaptive mechanism learns the optimum read reference voltage, which is expected to minimize errors when reading. We need to learn only one voltage per die, as all blocks within a die have similar aging properties because wear-leveling ensures that the blocks experience near-equal levels of wear-out. During the course of a day, the threshold voltage distribution shifts by only a small amount, and hence the read reference voltage stays close to its optimum value.

In our mechanism, the SSD controller first writes known, pseudo-random data to 100 test LSB pages (selected from 10 different blocks) within the flash die. These test pages are left in a partially-programmed state. Next, the controller uses the read-retry mechanism [5, 9, 40, 47] to read the test pages with the current read reference voltage and with the next four higher voltages. The higher voltages can compensate for the threshold voltage shifts that took place during the day, as program interference and read disturb can only increase the threshold voltage (see Section 3.2). The controller then compares the value it wrote to each test page (which it knows) with the value read from the flash chip, to count the number of read errors for each page at each voltage. Finally, the controller selects the voltage that generates the lowest number of errors as the optimum LSB read reference voltage.

Errors Removed. The adaptive read reference voltage mechanism only *mitigates* (but does not eliminate) the errors induced due to two-step programming. We perform experiments on real flash chips to characterize the error count reduction due to this mechanism. Figure 12 shows the reduction in error rate. For a new flash chip, without the effects of wear-out (i.e., it has only experienced *one* program/erase, or P/E, cycle), our adaptive read reference voltage mechanism lowers the raw bit error rate for reading partially-programmed wordlines by 32.7%. Even at 5K P/E cycles, after additional errors appear due to the effects of wear-out [9], our mechanism lowers the raw bit error rate by 21.0%.

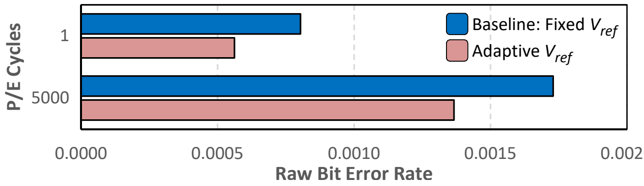


Figure 12: Reduction in raw bit error rate with our adaptive read reference voltage mechanism for partially-programmed pages, shown for two levels of wear-out.

Storage Overhead. Our mechanism requires only 1B of storage per flash die, which can represent up to 256 possible read reference voltages. For a 1TB SSD with 64 dies, this requires only 64B of storage.

Latency Overhead. To learn the read reference voltage for each die, the controller programs 100 LSB pages (1.6 ms each), and performs five reads (100 µs each) per page, taking a total of 210.0 ms. Even if we conservatively assume that learning is serialized for all dies, our algorithm requires 13.4 s each day for an SSD with 64 dies, which is negligible. Our mechanism does *not* add any latency to the read operation.

6.3. Multiple Pass-Through Voltages

The goal of our third mechanism is to minimize the errors that occur as a result of read disturb. As we discussed in Section 4.3, the number of errors grows significantly as the difference between the cell threshold voltage (V_{th}) and pass-through voltage (V_{pass}) increases. Figure 13a demonstrates how V_{pass} is currently applied to the unread cells within a block. Regardless of the programming status of a wordline, V_{pass} is set to be greater than the maximum possible cell threshold voltage. However, as discussed in Section 2, V_{th} on a partially-programmed wordline should not exceed *half* of the maximum threshold voltage, and the V_{th} of an unprogrammed cell is *close to 0V*. Thus, *for partially-programmed and unprogrammed wordlines, there is always a large gap between V_{pass} and V_{th}* (top of Figure 13a), *leading to a much larger number of* read disturb errors for these wordlines than for fully-programmed wordlines. If this gap can be minimized, the number of errors would decrease significantly. The key idea of our mechanism is to provide *three* different pass-through voltages for an open block, as shown in Figure 13b: one for fully-programmed wordlines (V_{pass}), another for partially-programmed wordlines ($V_{pass}^{partial}$), and a third for unprogrammed wordlines (V_{pass}^{erase}).

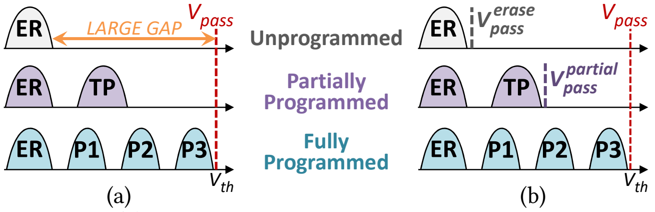


Figure 13: (a) Applying single V_{pass} to all unread wordlines; (b) Our multiple pass-through voltage mechanism, where different voltages are applied based on the the wordline's programming status, to minimize the effects of read disturb.

Mechanism. Instead of applying one pass-through voltage per block [10], we propose to modify the SSD so it applies different pass-through voltages onto wordlines based on their most recent programming step completed. Our mechanism ensures that the gap between the selected pass-through voltage and the highest possible threshold voltage for partially-programmed and unprogrammed cells is minimized.

Figure 13b illustrates how the three voltages are selected. All of an unprogrammed wordline's cells are in the ER state, as shown in the top of Figure 13b, so their threshold voltages should be close to 0V. For cells in an unprogrammed wordline, the controller uses V_{pass}^{erase}, which is set to slightly above 0V. None of a partially-programmed wordline's cell threshold voltages exceed the highest voltage of the TP state (see middle of Figure 13b). For cells in a partially-programmed wordline, the controller uses $V_{pass}^{partial}$, which is set to slightly higher than the highest voltage of the TP state. For a fully-programmed wordline, the controller uses the same pass-through voltage (V_{pass}) as the baseline (see bottom of Figure 13b).

Our mechanism does *not* require any adaptive learning, as we set one *fixed* value across the entire chip for each of our three pass-through voltages. The controller already knows the programming status of each wordline, as it tracks the next page to program, so no additional state tracking is required.

Errors Removed. We perform experiments on real flash chips to characterize the reduction in the raw bit error rate as a result of employing multiple pass-through voltages. As current SSDs do not allow the pass-through voltage to be changed, we use the read-retry mechanism [5, 9, 40, 47] to emulate lowering the pass-through voltage. In order to simulate N read disturbs, we first program known pseudo-random data to some of the pages in a block (leaving some wordlines partially programmed or unprogrammed), and then repeatedly try to "read" a single page in the block N times with the read reference voltage set to our desired V_{pass}. We then program the remaining pages within the block, perform a normal read of both the LSB and MSB pages on the selected wordline, and compare the data stored within the flash memory to the correct data, counting the number of errors.

Figure 14 shows the RBER when the SSD uses a single V_{pass} value (top), and when it uses multiple pass-through voltages (bottom), for a flash block that has endured 6K P/E cycles. We assume that ECC reaches its error correction capability at an RBER of 0.003. As we saw in Section 4.3, for wordlines that were fully programmed before read disturb occurs, the MSB pages have a high RBER under both mechanisms (the orange dashed line in Figure 14). As we also saw in Section 4.3, for wordlines partially programmed and unprogrammed before read disturb occurs, the LSB pages (the solid blue line in the top graph of Figure 14) have the highest RBER when using a single pass-through voltage. As a result, these pages limit the block to sustaining only 8K read disturbs. Figure 14 (bottom) shows that our mechanism significantly reduces the error rate. The RBER of the LSB pages in partially-programmed and unprogrammed wordlines (solid blue line in the bottom graph of Figure 14) drops so greatly that even after 50K read disturbs, these pages are far from approaching the maximum RBER correctable by ECC. The pages within the flash block can now sustain 3.5x more read disturbs than with a single V_{pass}, as the dominant number of read disturb errors now exist on MSB pages in fully-programmed wordlines.

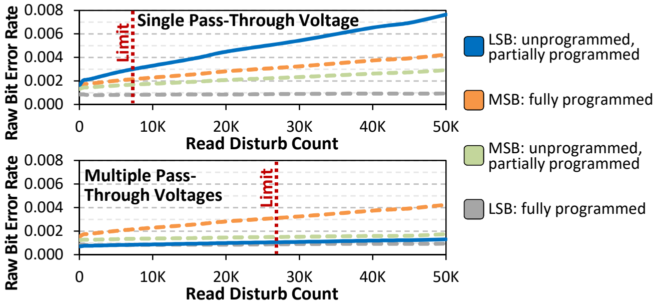


Figure 14: RBER when using a single baseline V_{pass} (top) vs. our multiple pass-through voltage mechanism (bottom) for different page types.

Figure 15 shows how the SSD lifetime changes (y-axis), with and without our mechanism, as the read disturb count per block varies (x-axis), for a fixed amount of ECC error correction capability (Section 3.1). As an SSD has a fixed error correction capability, it can only tolerate a maximum number of errors before exceeding the error correction capability. The lifetime of the SSD ends at the P/E cycle count when the error correction capability is exceeded. By reducing the read disturb count (i.e., reducing the number of read disturb errors that need to be corrected), our mechanism lets the SSD endure a higher number of P/E cycles before exhausting the error correction capability. Thus, our mechanism increases the SSD lifetime. For example, an SSD at 6K P/E cycles can sustain 5K read disturbs without our mechanism (as shown in Figure 15). Since our mechanism reduces the read disturb error count, it enables the SSD to sustain 5K read disturbs at 7K P/E cycles, thereby providing a 16% increase in lifetime.

Storage and Latency Overheads. Our mechanism uses the existing read-retry mechanism to adjust V_{pass} for each wordline within a block. Therefore, our mechanism does *not* require a new voltage generator, but simply needs an interface exposed by the SSD to set different values of V_{pass} (just like prior work has done [10]). As the three V_{pass} values remain constant throughout execution, no storage is required, and the mechanism does *not* incur any additional latency.

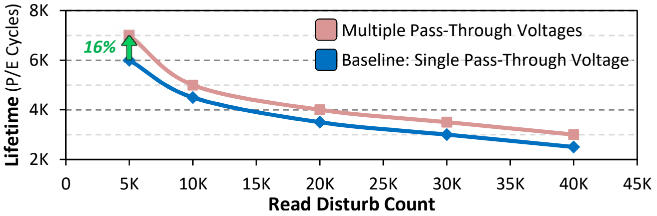


Figure 15: SSD lifetime as the read disturb count per block varies, with and without our multiple pass-through voltage mechanism.

7. Related Work

To our knowledge, this paper is the first to (1) experimentally characterize errors that occur due to the two-step programming method commonly used in MLC NAND flash memory; (2) reveal new reliability and security vulnerabilities exposed by two-step programming in flash memory; and (3) develop novel solutions to reduce these vulnerabilities. We briefly describe related works in various areas.

Flash Memory Error Characterization and Understanding. Prior works study various types of NAND flash memory errors derived from circuit-level noise, such as retention noise [3,4,6,7], read disturb noise [10], cell-to-cell program interference noise [3,5,7,8], and P/E cycling noise [3,7,9,31,38]. None of these works characterize how program interference and read disturb significantly increase errors within the unprogrammed or partially-programmed cells of an open block due to the vulnerabilities in two-step programming, nor do they develop mechanisms that exploit or mitigate such errors.

Program Interference Error Mitigation Mechanisms. Prior works model the behavior of program interference, and propose mechanisms that estimate the optimal read reference voltage once interference has occurred [5,8]. These works minimize program interference errors only for *fully-programmed* wordlines, by modeling the change in the threshold voltage distribution as a result of the interference. These models are fitted to the distributions of wordlines after *both* the LSB and MSB pages are programmed, and are unable to determine and mitigate the shift that occurs for wordlines that are *partially programmed*. In contrast, we propose mechanisms that specifically address the program interference resulting from two-step programming, and reduce the number of errors induced on LSB pages in *both* partially-programmed and unprogrammed wordlines.

Read Disturb Error Mitigation Mechanisms. One patent proposes a mechanism that uses counters to monitor the total number of reads to each block. Once a block's counter exceeds a threshold, the mechanism remaps and rewrites all of the valid pages within the block to remove the accumulated read disturb errors [18]. Another patent proposes to monitor the MSB page error rate to ensure that it does not exceed the ECC error correction capability, to avoid data loss [39]. Both of these mechanisms monitor pages *only* from *fully-programmed wordlines*. Unfortunately, as we observed (in Section 4.3), LSB pages in partially-programmed and unprogrammed wordlines are twice as susceptible to read disturb as pages in fully-programmed wordlines. If only the MSB page error rate is monitored, read disturb may be detected too late to correct some of the LSB pages. Another prior work dynamically changes the pass-through voltage for each block to reduce the impact of read disturb [10]. As a single voltage is applied to the whole block, this mechanism does *not* help significantly with the LSB pages in partially-programmed and unprogrammed wordlines. In contrast, our read disturb mitigation technique (Section 6.3) specifically targets these LSB pages by applying different pass-through voltages in an open block, optimized to the different programmed states of each wordline, to reduce read disturb errors.

Using Flash Memory for Security Applications. Some prior works studied how flash memory can be used to enhance the security of applications. One work uses flash memory as a secure channel to hide information, such as a secure key [45]. Other works use flash memory to generate random numbers and digital fingerprints [44, 46]. None of these works study vulnerabilities that exist within the flash memory.

Two-Step vs. One-Shot Programming. One-shot programming shifts flash cells directly from the erased state to their final target state in a single step. For smaller transistors with less distance between neighboring flash cells, such as those in sub-40nm 2D NAND flash memory, two-step programming has replaced one-shot programming to alleviate the coupling capacitance resulting from cell-to-cell program interference [37]. 3D NAND flash memory currently uses one-shot programming, as the chips use larger process technology nodes (>40nm). However, once the number of 3D-stacked layers reaches its upper limit, 3D NAND will scale to smaller transistors, and we expect that the increased program interference will again require two-step programming (just as it happened for 2D NAND in the past [24, 37]).

8. Conclusion

This paper shows that the two-step programming mechanism commonly employed in modern MLC NAND flash memory chips opens up new vulnerabilities to errors, based on an experimental characterization of modern 1X-nm MLC NAND flash chips. We show that the root cause of these vulnerabilities is the fact that when a partially-programmed cell is set to an intermediate threshold voltage, it is much more susceptible to both cell-to-cell program interference and read disturb. We demonstrate that (1) these vulnerabilities lead to errors that reduce the overall reliability of flash memory, and (2) attackers can potentially exploit these vulnerabilities to maliciously corrupt data belonging to other programs. Based on our experimental observations and the resulting understanding, we propose three new mechanisms that can remove or mitigate these vulnerabilities, by eliminating or reducing the errors introduced as a result of the two-step programming method. Our experimental evaluation shows that our new mechanisms are effective: they can either eliminate the vulnerabilities with modest/low latency overhead, or drastically reduce the vulnerabilities and reduce errors with negligible latency or storage overhead. We hope that the vulnerabilities we analyzed and exposed in this work, along with the experimental data we provided, open up new avenues for mitigation as well as for exposure of other potential vulnerabilities due to internal flash memory operation.

Acknowledgments

We thank the anonymous reviewers for their feedback. This work is partially supported by the Intel Science and Technology Center, the CMU Data Storage Systems Center, NSF grants 1212962/1320531, and gifts from Intel and Seagate.

References

[1] Y. Cai *et al.*, "Characterization and Mitigation of Reliability and Security Vulnerabilities in MLC NAND Flash Memory Programming," Carnegie Mellon Univ., SAFARI Research Group, Tech. Rep. 2017-002, 2017.

[2] Y. Cai *et al.*, "FPGA-Based Solid-State Drive Prototyping Platform," in *FCCM*, 2011.

[3] Y. Cai *et al.*, "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis," in *DATE*, 2012.

[4] Y. Cai *et al.*, "Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery," in *HPCA*, 2015.

[5] Y. Cai *et al.*, "Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation," in *ICCD*, 2013.

[6] Y. Cai *et al.*, "Flash Correct and Refresh: Retention Aware Management for Increased Lifetime," in *ICCD*, 2012.

[7] Y. Cai *et al.*, "Error Analysis and Retention-Aware Error Management for NAND Flash Memory," *Intel Technology Journal*, 2013.

[8] Y. Cai *et al.*, "Neighbor Cell Assisted Error Correction in MLC NAND Flash Memories," in *SIGMETRICS*, 2014.

[9] Y. Cai *et al.*, "Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling," in *DATE*, 2013.

[10] Y. Cai *et al.*, "Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery," in *DSN*, 2015.

[11] R. Cernea *et al.*, "A 34MB/s-Program-Throughput 16Gb MLC NAND with All-Bitline Architecture in 56nm," in *ISSCC*, 2008.

[12] R.-A. Cernea *et al.*, "A 34 MB/s MLC Write Throughput 16 Gb NAND with All Bit Line Architecture on 56 nm Technology," *JSSC*, 2009.

[13] J. Cha and S. Kang, "Data Randomization Scheme for Endurance Enhancement and Interference Mitigation of Multilevel Flash Memory Devices," *ETRI Journal*, 2013.

[14] K. Choi, "NAND Flash Memory," Samsung Electronics Co., Ltd., 2010.

[15] J. Cooke, "The Inconvenient Truths of NAND Flash Memory," in *Flash Memory Summit*, 2007.

[16] G. Dong *et al.*, "Using Data Postcompensation and Prediction to Tolerate Cell-to-Cell Interference in MLC NAND Flash Memory," *TCAS I*, 2010.

[17] R. H. Fowler and L. Nordheim, "Electron Emission in Intense Electric Fields," in *Proc. R. Soc. A*, 1928.

[18] H. H. Frost *et al.*, "Efficient Reduction of Read Disturb Errors in NAND Flash Memory," U.S. Patent No. 7,818,525. 2010.

[19] L. M. Grupp *et al.*, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *MICRO*, 2009.

[20] K. Ha *et al.*, "A Read-Disturb Management Technique for High-Density NAND Flash Memory," in *APSys*, 2013.

[21] T. Hara *et al.*, "A 146 mm^2 8 Gb NAND Flash Memory with 70 nm CMOS Technology," in *ISSCC*, 2005.

[22] C. Kim *et al.*, "A 21 nm High Performance 64 Gb MLC NAND Flash Memory with 400 MB/s Asynchronous Toggle DDR Interface," *JSSC*, 2012.

[23] J. Kim *et al.*, "Parameter-Aware I/O Management for Solid State Disks (SSDs)," *TC*, 2012.

[24] Y. S. Kim *et al.*, "New Scaling Limitation of the Floating Gate Cell in NAND Flash Memory," in *IRPS*, 2010.

[25] C. Lee *et al.*, "A 32-Gb MLC NAND Flash Memory with Vth Endurance Enhancing Schemes in 32 nm CMOS," *JSSC*, 2011.

[26] D.-H. Lee and W. Sung, "Least Squares Based Cell-to-Cell Interference Cancelation Technique for Multi-Level Cell NAND Flash Memory," in *ICASSP*, 2012.

[27] J.-D. Lee *et al.*, "Effects of Floating-Gate Interference on NAND Flash Memory Cell Operation," *EDL*, 2002.

[28] J. T. Lin *et al.*, "System, Method and Memory Device Providing Data Scrambling Compatible with On-Chip Copy Operation," U.S. Patent No. 8,301,912. 2012.

[29] R.-S. Liu *et al.*, "DuraCache: A Durable SSD Cache Using MLC NAND Flash," in *DAC*, 2013.

[30] R.-S. Liu *et al.*, "Optimizing NAND Flash-Based SSDs via Retention Relaxation," in *FAST*, 2012.

[31] Y. Luo *et al.*, "Enabling Accurate and Practical Online Flash Channel Modeling for Modern MLC NAND Flash Memory," *JSAC*, 2016.

[32] J. Meza *et al.*, "A Large-Scale Study of Flash Memory Errors in the Field," in *SIGMETRICS*, 2015.

[33] N. Mielke *et al.*, "Bit Error Rate in NAND Flash Memories," in *IRPS*, 2008.

[34] V. Mohan *et al.*, "reFresh SSDs: Enabling High Endurance, Low Cost Flash in Datacenters," Univ. of Virginia, Tech. Rep. CS-2012-05, 2012.

[35] Open NAND Flash Interface, "ONFI 4.0 Specifications," http://www.onfi.org/specifications, 2015.

[36] Y. Pan *et al.*, "Quasi-Nonvolatile SSD: Trading Flash Memory Nonvolatility to Improve Storage System Performance for Enterprise Applications," in *HPCA*, 2012.

[37] K.-T. Park *et al.*, "A Zeroing Cell-to-Cell Interference Page Architecture with Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories," *JSSC*, 2008.

[38] T. Parnell *et al.*, "Modelling of the Threshold Voltage Distributions of Sub-20nm NAND Flash Memory," in *GLOBECOM*, 2014.

[39] A. Schushan, "Refreshing of Memory Blocks Using Adaptive Read Disturb Threshold," U.S. Patent Appl. No. 20140173239. 2014.

[40] H. Shim *et al.*, "Highly Reliable 26nm 64Gb MLC E2NAND (Embedded-ECC & Enhanced-Efficiency) Flash Memory with MSP (Memory Signal Processing) Controller," in *VLSIT*, 2011.

[41] K.-D. Suh *et al.*, "A 3.3V 32 Mb NAND Flash Memory with Incremental Step Pulse Programming Scheme," *JSSC 1995*.

[42] K. Takeuchi *et al.*, "A Negative Vth Cell Architecture for Highly Scalable, Excellently Noise-Immune, and Highly Reliable NAND Flash Memories," *JSSC*, 1999.

[43] J. P. van Zandwijk, "A Mathematical Approach to NAND Flash-Memory Descrambling and Decoding," *Digital Investigation*, 2015.

[44] Y. Wang *et al.*, "Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints," in *SP*, 2012.

[45] Y. Wang *et al.*, "Hiding Information in Flash Memory," in *SP*, 2013.

[46] S. Q. Xu *et al.*, "Understanding Sources of Variations in Flash Memory for Physical Unclonable Functions," in *IMW*, 2014.

[47] L. Zuolo *et al.*, "SSDExplorer: A Virtual Platform for Performance/Reliability-Oriented Fine-Grained Design Space Exploration of Solid State Drives," *TCAD*, 2015.