FLASH MEMORY FOR UBIQUITOUS HARDWARE SECURITY FUNCTIONS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

M.S.

by

Yinglei Wang

December 2013

ABSTRACT

We demonstrate that unmodified commercial Flash memory can provide three important security functions: true random number generation, digital fingerprinting and information hiding. Taking advantage of random telegraph noise (a type of quantum noise source in highly scaled Flash memory cells) enables high quality true random number generation at a rate up to 10Kbits / second. A scheme based on partial programming exploits process variation in threshold voltages to allow quick generation of many unique fingerprints that can be used for identification and authentication. Information hiding hides data within an analog characteristic of Flash, the program time of individual bits. Because the technique uses analog behaviors, normal Flash memory operations are not affected and hidden information is invisible in the data stored in the memory. Even if an attacker checks a Flash chip's analog characteristics, experimental results indicate that the hidden information is difficult to distinguish from inherent manufacturing variation or normal wear on the device. Moreover, the hidden data can survive erasure of the Flash memory data. All schemes require no change to Flash chips or interfaces, and do not require additional hardware.

BIOGRAPHICAL SKETCH

Yinglei Wang attended Peking(Beijing) University in China as an EECS undergraduate student from year 2006 to 2010. She entered the MS/PHD program in ECE, Cornell University in Fall, 2010. She worked with Prof. Edwin C. Kan, Prof. G. Edward Suh and Prof. Christopher Batten on flash memory security and on-chip network. She will work as a software engineer at Oracle from January, 2014.

This document is dedicated to my parents and my husband Xuetian Weng,

who is now a PHD student in Computer Science.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

Flash memory has gained a ubiquitous place in the computing landscape today. Virtually all mobile devices such as smartphones and tablets rely on Flash memory as their non-volatile storage. Flash memory is also moving into laptop and desktop computers, intending to replace the mechanical hard drive. Floating-gate non-volatile memory is even more broadly used in electronic applications with a small amount of non-volatile memory. For example, even 8-bit or 16-bit microcontrollers for embedded systems commonly have on-chip EEPROMs to store instructions and data. Many people also carry Flash memory as stan-dalone storage medium as in USB memory sticks and SD cards.

In this paper, we propose to utilize analog behaviors of off-the-shelf Flash memory to enable hardware-based security functions in a wide range of electronic devices without requiring custom hardware. More specifically, we show that a standard Flash memory interface can be used to generate true random numbers from quantum and thermal noises and to produce device fingerprints based on manufacturing variations. This paper also introduces a technique to hide information in analog characteristics of Flash memory in a way that the hidden bits are not visible at all from the viewpoint of normal Flash memory content. Our technique encodes a hidden bit in the program time of a group of Flash cells; a fast program time encodes bit '1' and a slow program time encodes bit '0'. We found that writing 0 into a Flash cell incurs more stress on the cell than writing 1, which in turn results in a larger decrease in the program time

of the corresponding cell. While the program time of individual cells cannot be accurately controlled, our experiments demonstrate that bits can be reliably encoded in the program time using many cells collectively. The techniques can be applied to any floating-gate non-volatile memory in general, and does not require any hardware modifications to todays Flash memory chips, allowing them to be widely deployed.

1.2 Quantum Random Number Generation

Hardware random number generators (RNGs) provides important foundations in building secure systems. For example, true randomness is a critical ingredient in many cryptographic primitives and security protocols; random numbers are often required to generate secret keys or prevent replays in communications. While pseudo-random number generators are often used in todays systems, they cannot provide true randomness if a seed is reused or predictable. As an example, a recent study showed that reuse of virtual machine (VM) snapshots can break the Transport Level Security (TLS) protocol due to predictable random numbers [1]. Given the importance of a good source of randomness, high security systems typically rely on hardware RNGs. Instead of requiring custom hardware modules for RNGs, we found that analog noise in Flash memory bits can be used to reliably generate true random numbers. An interesting finding is that the standard Flash chip interface can be used to put a memory bit in partially programmed state so that the internal noise can be observed through the digital interface. There exist two sources of true randomness in Flash bits, Random Telegraph Noise (RTN) and thermal noise. While both sources can be leveraged for RNGs, our scheme focuses on RTN, which is quantum noise. Un-

2

like thermal noise, which can be reduced significantly at extremely low temperatures, RTN behavior continues at all temperature ranges. Moreover, the quantum uncertainty nature of RTN provides a better entropy source than system level noises which rely on the difficulty of modeling complex yet deterministic systems. Our algorithm automatically selects bits with RTN behavior and converts RTN into random binary bits. Experimental results demonstrate that the RTN behavior exists in Flash memory and can be converted into random numbers through the standard Flash interface. The Flash-based RNG is tested using the NIST test suite [2] and is shown to pass all tests successfully. Moreover, we found that the RNG works even at a very low temperature (-80 C). In fact, the RTN behavior is more visible at low temperatures. On our test platform, the Flash RNG generates about 1K to 10K bits per second. Overall, the experiments show that true random numbers can be generated reliably from off-the-shelf Flash memory chips without requiring custom circuits.

1.3 Device Fingerprint

In addition to generating true random numbers, we also found that the standard Flash interface can be used to extract fingerprints (or signatures) that are unique for each Flash chip. For this purpose, our technique exploits inherent random variations during Flash manufacturing processes. More specifically, we show that the distributions of transistor threshold voltages can be measured through the standard Flash interface using incremental partial programming. Experimental results show that these threshold voltage distributions can be used as fingerprints, as they are significantly different from chip to chip, or even from location to location within a chip. The distributions also stay relatively stable

across temperature ranges and over time. Thanks to the large number of bits (often several gigabits) in modern Flash chips, this technique can generate a large number of independent fingerprints from each chip.

The Flash fingerprints provide an attractive way to identify and/or authenticate hardware devices and generate device-specific keys, especially when no cryptographic module is available or a large number of independent keys are desired. For example, at a hardware component level, the fingerprints can be used to distinguish genuine parts from counterfeit components without requiring cryptography to be added to each component. The fingerprinting technique can also be used for other authentication applications such as turning a Flash device into a two-factor authentication token, or identifying individual nodes in sensor networks.

While the notion of exploiting manufacturing process variations to generate silicon device fingerprints and secret keys is not new and has been extensively studied under the name of Physical Unclonable Functions (PUFs) [3], the Flash-based technique in this paper represents a unique contribution in terms of its practical applicability. Similar to true RNGs, most PUF designs require custom circuits to convert unique analog characteristics into digital bits. On the other hand, our technique can be applied to off-the-shelf Flash without hardware changes. Researchers have recently proposed techniques to exploit existing bi-stable storage elements such as SRAMs [4] or Flash cells [5] to generate device fingerprints. Unfortunately, obtaining fingerprints from bi-stable elements requires a power cycle (power off and power on) of a device for every fingerprint generation. The previous approach to fingerprinting Flash only works for a certain types of Flash chips and takes long time (100 seconds for one

fingerprint) because it relies on rare errors called program disturbs. As an example, we did not see any program disturbs in SLC Flash chips that we used in experiments. To the best of our knowledge, the proposed device fingerprinting techniques is the first that is fast (less than 1 second for a 1024-bit fingerprint) and widely applicable without interfering with normal operation or requiring custom hardware.

1.4   Information Hiding

This part introduces a technique to hide information in analog characteristics of Flash memory in a way that the hidden bits are not visible at all from the viewpoint of normal Flash memory content. More specifically, our technique encodes a hidden bit in the program time of a group of Flash cells; a fast program time encodes bit '1' and a slow program time encodes bit '0'. We found that writing 0 into a Flash cell incurs more stress on the cell than writing 1, which in turn results in a larger decrease in the program time of the corresponding cell. While the program time of individual cells cannot be accurately controlled, our experiments demonstrate that bits can be reliably encoded in the program time using many cells collectively.

While a number of steganography techniques have been developed previously, our Flash-based technique provides unique benefits compared to typical digital steganography schemes where information is hidden in another form of digital content such as images and documents. In particular, the hidden information in Flash memory is decoupled from the Flash memory content and instead tied to the physical object. The following summarizes the main benefits

5

of our scheme compared to digital steganography.

- **Covert**: The proposed technique does not change normal Flash operations or content at all. As a result, inspecting the Flash memory content does not reveal any hidden information. All Flash memory operations can still be performed without any change, even with hidden information. In fact, our experimental results suggest that even analog characteristics of Flash memory such as page program/erase time do not change noticeably.

- **Erase tolerant**: The hidden information in Flash memory remains intact even if the entire Flash memory is erased and programmed with new content. In fact, our experiments show that the hidden information can survive even hundreds of program/erase operations.

- **Copy tolerant**: In typical digital steganography, the cover text with hidden information can be easily copied and stored so that it can be analyzed over time. The hidden information in our technique, however, is tied to physical Flash memory and can only be accessed by measuring the program time of individual memory cells while the Flash memory is in one's possession. Because modern Flash memory chips often contain tens or hundreds of billions of memory cells, fully characterizing a Flash chip without knowing the location of hidden bits is quite time consuming.

In a sense, the proposed information hiding technique is similar to physical steganography methods where information is hidden in physical objects. For example, people have used secret inks to write messages on blank parts of other messages [6]. However, the proposed technique provides a couple of key benefits over traditional physical steganography methods thanks to being electrical.

- **No hardware modification**: The proposed technique works on unmodified Flash chips using the standard interface. In fact, the technique can be implemented as a software program as long as a low-level Flash interface is exposed.

- **High capacity**: Thanks to the high capacity of Flash memory, our technique provides a fairly high capacity compared to traditional physical steganography techniques. For example, even if we hide one bit for every 512 Flash cells, a 8GB Flash chip can contain 16MB of hidden information.

Given the ubiquity of Flash memory and the easy applicability of the proposed scheme on commercial Flash chips, we believe that the technique can enable a number of interesting applications. An obvious application of the information hiding in Flash is a secure and covert storage of data. For example, a user can hide sensitive information in the Flash memory of a smartphone with confidence that others cannot retrieve the information even when the phone is lost or stolen. Information hiding provides an additional layer of protection on top of typical encryption by preventing an adversary from reading or even copying the ciphertext.

On the other hand, the capability to covertly communicate may be misused to bypass legitimate access control policies. For example, in the business world, the hidden information in Flash may be misused to export trade secrets. In this sense, this study points out the potential danger.

Another traditional application of information hiding is watermarking. In particular, given that the hidden information is tied to a physical Flash memory chip, the proposed technique can be used to embed watermarking in devices with Flash memory. For example, mobile or embedded devices may be water-

marked to help retrieve them when lost or stolen. Similarly, the watermarks can be used to distinguish genuine devices from low-quality counterfeits.

CHAPTER 2

FLASH MEMORY BACKGROUND

This section provides background material on Flash memory and its operating principles to aid understanding of our Flash-based information hiding scheme.

2.1 Floating Gate Transistors

Flash memory is composed of arrays of floating-gate transistors. A floating-gate transistor is a transistor with two gates, stacked on top of each other. One gate is electrically insulated (floating). Figure 2.1 shows an example of a floating-gate device. The control gate is on top. An insulated conductor, surrounded by oxide, is between the control gate and the channel. This conductor is the floating gate. Information is stored as the presence or absence of trapped charge on the floating gate. The trapped negative charge reduces the current flowing through the channel when the N-type MOS transistor is on. This current difference is sensed and translated into the appropriate binary value.

Flash cells without charge on their floating-gate allow full current flow in the channel and hence are read as a binary "1". The presence of charge on the floating-gate will discourage the presence of current in the channel, making the cell store a "0". Effectively, the charge on the floating-gate increases the threshold voltage (Vth) of a transistor. Single-level cells (SLC) store one bit of information per cell by using two threshold voltage levels. Multi-level cells (MLC) store more than one bit by more finely dividing the threshold voltage levels: for example, four levels can be used to store two bits per cell.
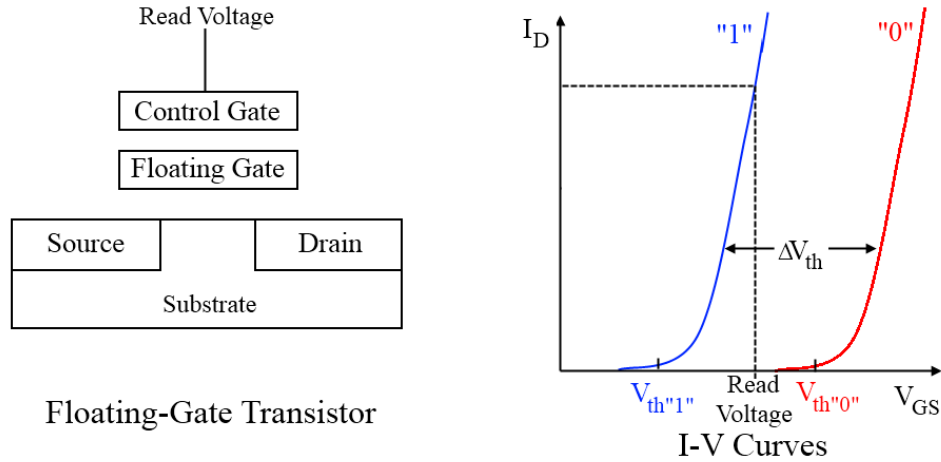
Figure 2.1: Flash memory cell based on a floating gate transistor.

2.2 Flash Organization and Operation

At a high-level, Flash memory provides three major operations: read, erase, and program (write). In order to read a bit in a Flash cell, the corresponding transistor is turned on and the amount of current is detected. A write to a Flash cell involves two steps. First, an erase operation pushes charge off the floating-gate by applying a large negative voltage on the control gate. Then, a program (write) operation stores charge on the floating-gate by selectively applying a large positive voltage if the bit needs to be zero.

An important concept in Flash memory operation is that of pages and blocks. Pages are the smallest unit in which data is read or written, and are usually 2KB to 8KB. Blocks are the smallest unit for an erase operation and made up of several pages, usually 32 - 128 pages. Note that Flash does not provide bit-level program or erase. To read an address from a Flash chip, the page containing the address is read. To update a value, the block that includes the address must be first erased. Then, the corresponding page is written with an update and other

pages in the block are restored.

2.3   Aging

Flash requires high voltages to store and erase information. The voltages involved place great stress on the device oxide; each program operation and each erase operation slightly damages the oxide, wearing out the device. After thousands of program and erase cycles, the oxide could have sustained enough damage to render the bit non-operational, leaving it in a stuck-at state or in a leaky state that cannot reliably hold information over a period of time. Flash is usually guaranteed by the manufacturer up to a certain number of program and erase cycles.

Even before failures, the stress causes the cell's analog characteristics to change. In particular, the program time that is required to flip a state from '1' to '0' for a cell tends to reduce as the number of program/erase (PE) cycles increases for that cell. We exploit this program time shift in order to hide information.

2.4   Partial Programming

Our information hiding scheme relies on the measurement of program time, the time it takes to program a Flash cell, at individual cell granularity. However, the standard Flash memory interface requires all bits in a page to be programmed together. Normally, a program operation on a page is held for a long enough

time that any cell level variation within a page is overcome. Therefore, the normal program time only reveals how long programming the entire page takes, not how long it takes to program individual bits.

To find the program time on a per-cell basis, we use a technique called "partial programming" [7]. The standard Flash memory interfaces allow the "partial program" of a cell by aborting a program operation before completion. If the program operation is interrupted, the Flash cell may be in an unreliable state that could be interpreted as 1 or 0. Further "partial programs" will accumulate charge on the floating gate and eventually result in the cell entering a stable programmed state, as if a full program was applied. Effectively, the number of partial program operations to flip a bit from 1 to 0 represents the program time for the bit. In this sense, we use the "partial programming" technique to to find program time for individual cells. After a partial program to a page, we read the page and record the state of each bit. When a bit changes to the programmed state (from 1 to 0), we note the number of partial programs required to flip the bit as the bit's program time.

CHAPTER 3

RANDOM NUMBER GENERATION

3.1 Theory and Implementation

3.1.1 Random Telegraph Noise (RTN)

The proposed RNG uses a device effect called Random Telegraph Noise (RTN) as the source of randomness. In general, RTN refers to the alternating capture and emission of carriers at a defect site (trap) of a very small electronic device, which generates discrete variation in the channel current [8]. The capture and emission times are random and exponentially distributed. RTN behavior can be distinguished from other noise using the power spectrum density (PSD), which is flat at low frequencies and $1/f2$ at high frequencies. In Flash memory, the defects that cause RTN are located in the tunnel-oxide near the substrate. The RTN amplitude is inversely proportional to the gate area and nearly temperature independent. As Flash memory cells shrink, RTN effects become relatively stronger and their impact on the threshold distribution of Flash memory cells, especially for multi-level cells, can be significant. Because RTN can be a major factor in Flash memory reliability, there have been a large number of recent studies on RTN in Flash memory from a reliability perspective [9, 10, 11]. While RTN is a challenge to overcome from the perspective of Flash memory operations, it can be an ideal source of randomness. RTN is caused by the capture and emission of an electron at a single trap, and is a physical phenomenon with random quantum properties. Quantum noise can be seen as the gold-standard for random number generation because the output of quantum events cannot
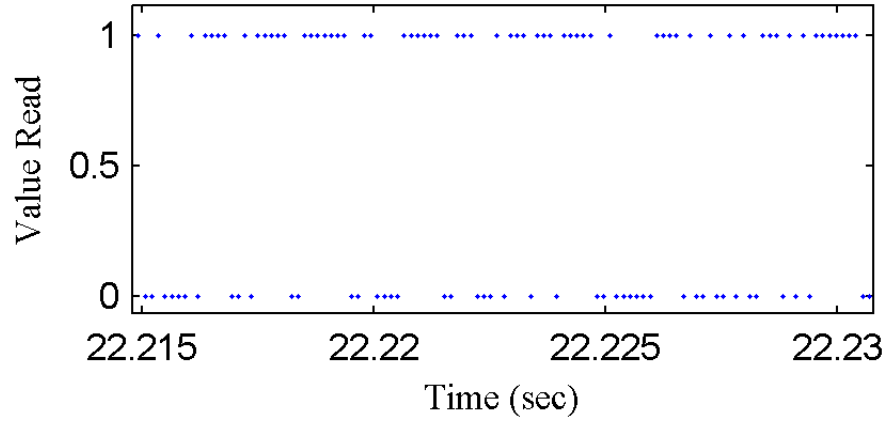
Figure 3.1: Thermal noise in Flash memory (time domain).

be predicted. As Flash memory cells scale to smaller technology nodes, the RTN effect will become stronger. Moreover, RTN behavior will still exist with increasing process variation and at extremely low temperatures.

3.1.2   Noise Extraction from Digital Interface

As digital devices, Flash memory is designed to tolerate analog noise; noise should not affect normal memory operations. In order to observe the noise for random number generation, a Flash cell needs to be in an unreliable state between well-defined erase and program states. Interestingly, we found that Flash cells can be put into the in-between state using the standard digital interface. In a high level, the approach first erases a page, issues a program command, and then issues a reset command after an appropriate time period to abort the program. This procedure leaves a page partially programmed so that noise can affect digital outputs. We found that the outcome of continuously reading a partially programmed bit oscillates between 1 and 0 due to noise.

14

For Flash memory in practice, experiments show that two types of noise co-exist: thermal noise and RTN. Thermal noise is white noise that exists in nearly all electronic devices. RTN can be observed only if a surface trap exists, the RTN amplitude is larger than that of thermal noise, and the sampling frequency (speed for continuous reads) is high enough. If any of these three conditions is not satisfied, only thermal noise will be observed as in Figure 3.1. In the case of thermal noise, a bit oscillates between the two states quickly, and the power spectral density (PSD) indicates white noise.

In the case that the RTN amplitude is comparable to thermal noise, a combination of RTN and thermal noise is observed as shown in Figure 3.2. This is reflected by the density change of 1s in the continuous reading. A moving average on the time domain helps to visualize the density change. The PSD of the result shows $1/f^2$ spectrum at low frequencies and becomes flat at high frequencies.

In some cases, the RTN amplitude is very high and dominates thermal noise. As a result, only RTN behaviors are visible through digital interfaces for these bits. As shown in Figure 3.3, continuous reads show clear clusters of 1s and 0s in the time domain. The power spectral density (PSD) of these bit sequences shows a clear RTN pattern of $1/f^2$.

For a bit with nearly pure RTN behavior, we further validated that the error pattern corresponds to RTN by plotting the distributions of up and down periods. As shown in Figure 3.4, both up time and down time nicely fit an exponential distribution as expected. Overall, our experiments show that both RTN and thermal noise exist in Flash memory and can be observed through a digital interface. While both noise types can be used for random number generation,

15
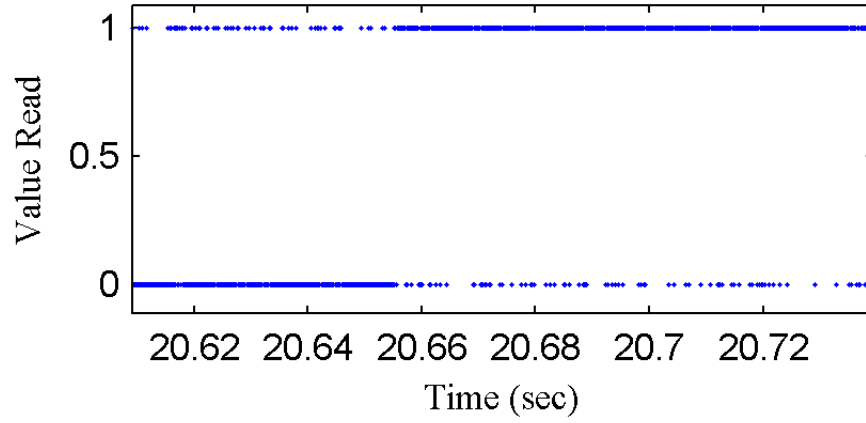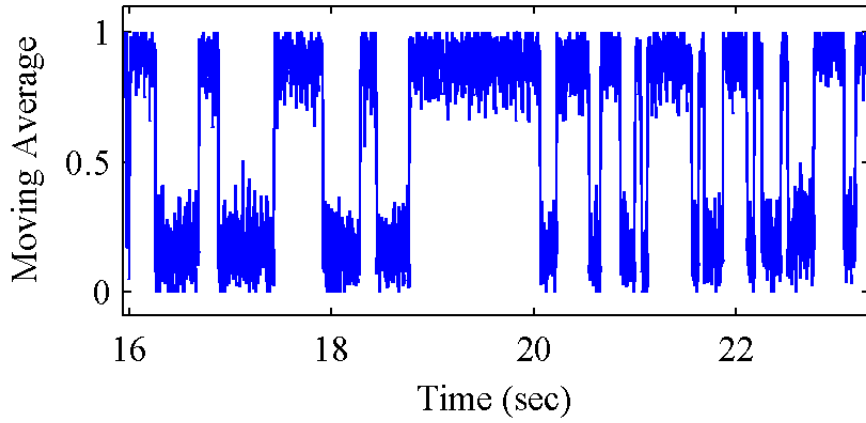
Figure 3.2: RTN with thermal noise in Flash memory. (a) Time domain. (b) Moving average of 29 points on the time domain.

we focus on RTN, which is more robust to temperature changes.

3.1.3 Random Number Generation Algorithms

In Flash memory devices, RTN manifests as random switching between the erased state (consecutive 1s) and programmed state (consecutive 0s). At a high-level, our Flash random number generator (RNG) identifies bits with RTN be-
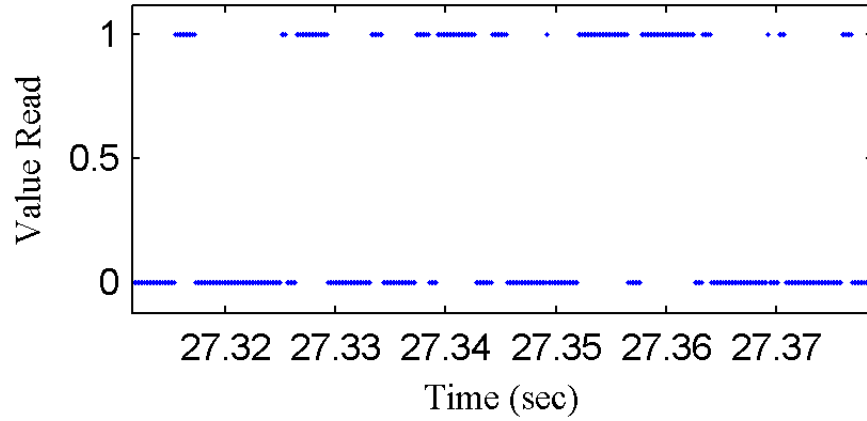
Figure 3.3: RTN in Flash memory (time domain).

havior, either pure RTN or RTN combined with thermal noise, and uses a sequence of time in the erased state (called up-time) and the time in the programmed state (called down-time) from those bits. In order to produce random binary outputs, the RNG converts the up-time and down-time sequence into a binary number sequence, and applies the von Neumann extractor for debiasing. We found that thermal noise itself is random and does not need to be filtered out.

Algorithm I shows the overall RNG algorithm. To generate random numbers from RTN, the first step is to identify bits with RTN or both RTN and thermal noise. To do this, one block in Flash memory is erased and then multiple incomplete programs with the duration of T are applied. After each partial program, a part of the page is continuously read N times and the outcome is recorded for each bit. In our experiments, we chose to read the first 80 bits (10 bytes) in a page for 1,000 times. For each bit that has not been selected yet, the algorithm checks if RTN exists using CheckRTN() and marks the bit location if there is RTN. As an optimization, the algorithm also records the number of partial programs when a bit is selected. The algorithm repeats the process until all bits

17
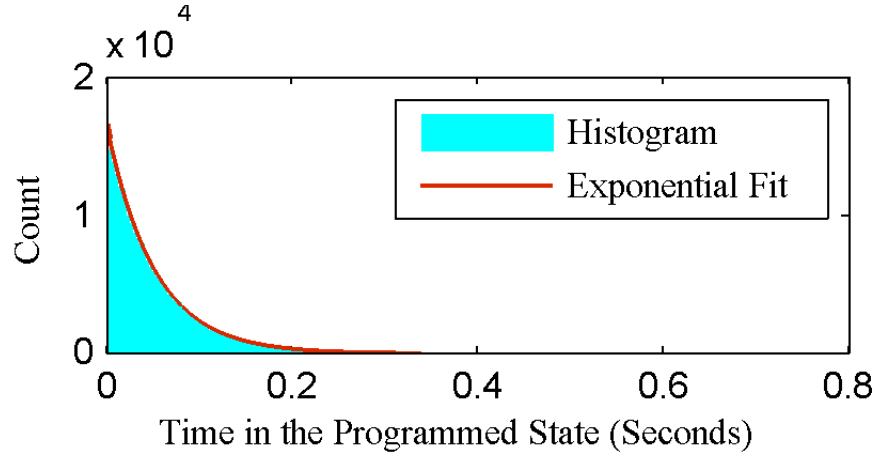
(a)
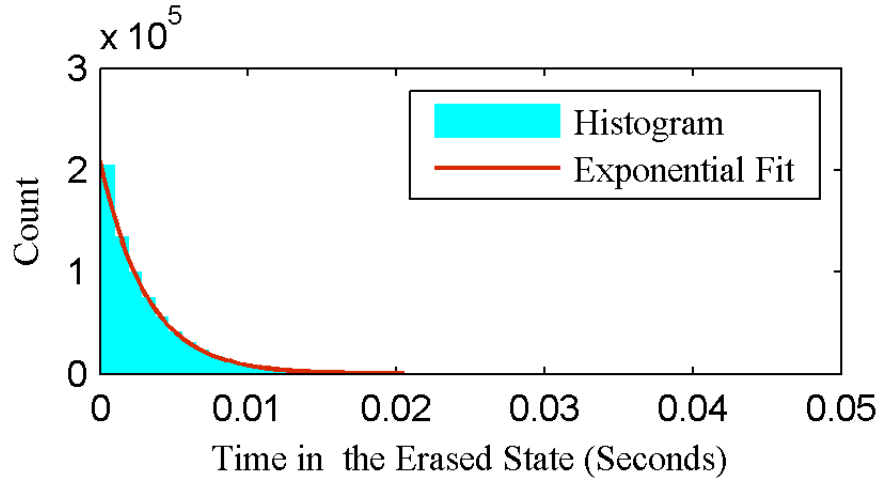
(b)

Figure 3.4: (a) Distribution of time in the programmed state. (b) Distribution of time in the erased state.

are checked for RTN. The second step is to partially program all of the selected bits to an appropriate level so that they will show RTN behavior. Finally, the algorithm reads the selected bits M times, records a sequence of up-time and down-time for each bit, and converts the raw data to a binary sequence.

The function CheckRTN() in Algorithm II determines whether there is RTN in a bit based on a trace from N reads. The algorithm first filters out bits that

Algorithm I Overall Flash RNG algorithm

Erase a block;

Num = 0;
do
    Partially program a page for T;
    Num++;

    Read Nbytes in a page N times, and record a
    trace for each bit trace[bit];
    For each bit in Nbytes, not selected yet
        If (CheckRTN(trace[bit]) == true)
            Selected[bit] = yes;
            NumProgram[bit] = Num;
    End for
    repeat until most bits are programmed.

ProgramSelectBits(Selected);

Read selected bits M times, and record up-time and down-time;
For each bit
     ConvertToBinary(rawdata);
End for

Figure 3.5: Overall Flash RNG algorithm

Algorithm II Determine whether there is RTN in a bit

If trace[bit] has over 98% 1/0s
    Return false;
End if

Calculate the power spectrum density (PSD);
Convert PSD to the log scale in both x-y;
If PSD slope is always $< T_{slope}$ for all high frequency ($> T_{freq}$)
    Return RTN
End if

If PSD slope is $< T_{slope}$ at least one interval (Invl) at a high frequency ($> T_{freq}$)
    Return RTN-Thermal
End if

Figure 3.6: Determine whether there is RTN in a bit

```
┌─────────────────────────────────────────────────┐
│ Algorithm III Program selected bits to proper levels │
│ where RTN could be observed.                    │
│                                                 │
│ For each selected bit                           │
│     Do (NumProgram[bit]-K) partial programs;    │
│                                                 │
│     do {                                        │
│        Partially program the bit for T;         │
│                                                 │
│        Read the bit N times;                    │
│        Find Max and Min for moving averages;    │
│                                                 │
│        If Max > TMax and Min < TMin             │
│           Break;                                │
│        End if                                   │
│     } repeat up to L times                      │
│ End for                                         │
└─────────────────────────────────────────────────┘
```

Figure 3.7: Program selected bits to proper levels where RTN could be ob-
served.

almost always (more than 98%) produce one result, either 1 or 0. For the bits
with enough noise, the algorithm uses the power spectral density (PSD) to dis-
tinguish RTN from thermal noise; PSD for RTN has a form of $1/f^2$ at a high
frequency. To check this condition, the algorithm computes the PSD, and con-
verts it to a log-scale in both x and y axes. If the result has a slope less than
Tslope (we use -1.5, the ideal value is -2) for all frequencies higher than Tfreq
(we use 200Hz), the algorithm categorizes the bit as RTN only. If the PSD has a
slope less than Tslope for any interval larger than than Invl (we use 0.2) at a high
frequency, the bit is categorized as a combination of RTN and thermal noise.

The function ProgramSelectBits() in Algorithm III programs selected bits to
a proper level where RTN can be observed. Essentially, the algorithm aims to
take each bit to the point near where they were identified to have RTN. The
number of partial programs that were required to reach this point before were

Algorithm IV Convert the raw data to binary random sequence.

If the bit has both RTN and thermal noise
   For each up/down-time in raw data
      Output = LSB(up/down-time);
   End for
End if

If the bit has only RTN
   do {
      For each up/down-time in raw data
         Output = LSB(up/down-time);
         Shift right up/down-time by one bit;
      End for
   } repeat until all up/down time are zero;
End if

Perform von Neumann de-biasing

Figure 3.8: Convert the raw data to binary random sequence.

recorded in NumProgram[Bit]. For each selected bit, the algorithm first performs partial programs with the duration of T based on the number recorded earlier (NumProgram[Bit]-K). Then, the algorithm performs up to L more partial program operations until a bit shows RTN behavior. The RTN behavior is checked by reading the bit N times, and see if the maximum of moving averages is greater than a threshold ($TMax = 0.7$) and the minimum is less than another threshold ($TMin = 0.3$).

Finally, the function ConvertToBinary() converts the raw data to a binary random sequence. For bits with both RTN and thermal noise, the up-time and down-time tend to be short. So only the LSBs of these numbers are used. Essentially, for every up-time and down-time, the algorithm produces 1 if the time is odd and 0 otherwise. Effectively, this is an even-odd scheme. For bits with perfect RTN behavior, up-time and down-time tend to be longer and we use

more LSBs from the recorded up/down-time. In this case, we first produce a bit based on the LSB, then the second LSB, the third LSB, and so on until all extracted bits become 0. Finally, for both methods, we apply the von Neumann de-biasing method. The method takes two bits at a time, throws away both bits if they are identical, and takes the first bit if different. This process is described in Algorithm IV.

The stability of the bits in the partially programmed state is also important. We define the stability as how long a bit stays in the partially programmed state where RTN behavior can be observed. This is determined by the retention time of the Flash memory chip and the amplitude of the RTN compared to the designed noise margin. Assume the amplitude of the RTN is Ar, the noise margin of Flash memory is An, and the Flash retention time is 10 year, then the stable time for random number generation after partial programming will be roughly $Ts = Ar/An * 10$ years. This means that after time Ts, a bit needs to be reset and reprogrammed. In our experiments, the bit that is shown in Figure 3.4 was still showing ideal RTN behavior even after 12 hours.

3.2   Experimental Results

This section presents evaluation results for the random number generation techniques for Flash memory devices. The two main metrics for random number generation are randomness and throughput. For security, the RNG must be able to reliably generate true random numbers across a range of environmental conditions over time. For performance, higher throughput will be desirable.
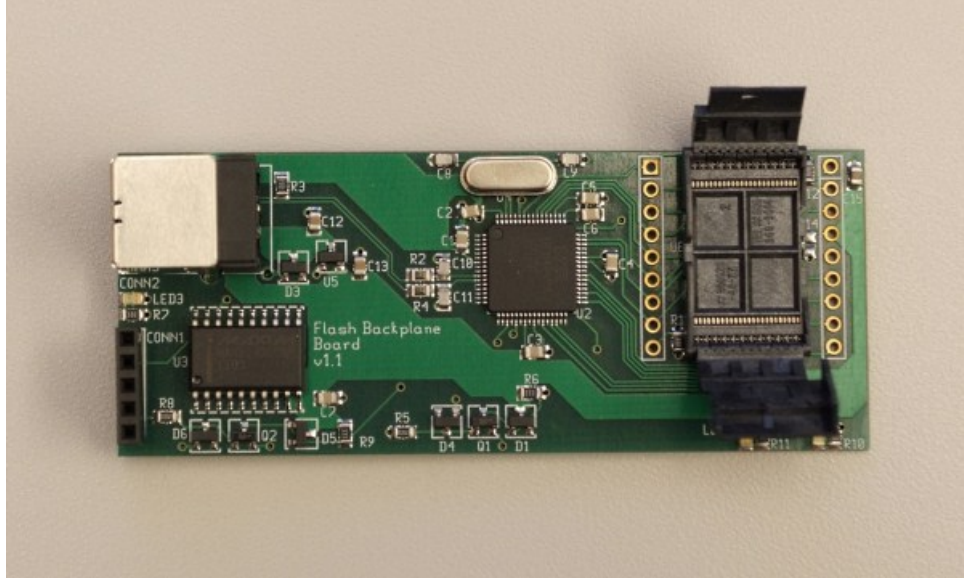
Figure 3.9: Flash test board.

3.2.1 Evaluation Setup

Our experiments use a custom Flash test board as shown in Figure 5.8. The board is made entirely with commercial off-the-shelf (COTS) components with a custom PCB. There is a socket to hold a Flash chip under test, an ARM microprocessor to issue commands and receive data from the Flash chip, and a Maxim MAX-3233 chip to provide a serial (RS-232) interface. USB support is integrated into the ARM microcontroller. We also wrote the code to test the device. The setup represents typical small embedded platforms such as USB Flash drives, sensor nodes, etc. This device shows that the techniques can be applied to commercial off-the-shelf devices with no custom integrated circuits (ICs).

The experiments in this paper were performed with four types of Flash memory chips from Numonyx, Micron and Hynix, as shown in Table 5.1.

23

Manufacturer	Part Number	Size	Qty	Process
Hynix	HY27UF084G2B	4 Gbit	10	5xnm class SLC
Micron	MT29F2G08ABA EAWP-IT:E4	2 Gbit	24	34nm SLC
Micron	MT29F16G08CB ACAWP:C	16 Gbit	5	– MLC
Numonyx	NAND04GW 3B2DN6	4 Gbit	3	57nm SLC

Table 3.1: Tested Flash chips.

3.2.2 Randomness

Historically, three main randomness test suites exist. The first one is from Donald Knuths book The Art of computer Programming (1st edition, 1969) [12] which is the most quoted reference in statistical testing for RNGs in literature. Although it was a standard for many decades, it appears to be outdated in todays view and it allows many bad generators to pass the tests. The second one is the diehard test suite from Florida State University. The test suite is stringent in the sense that they are difficult to pass. However, the suite has not been maintained in recent years. Therefore, it was not selected as the tests for this study. The third one is developed by National Institute of Standards and Technology (NIST) which is a measurement standard laboratory and a non-regulatory agency of the United States Department of Commerce. The NIST Statistical Test Suite is a package consisting of 15 tests that were developed to test the randomness of arbitrary long binary sequences produced by either hardware or software. The test suite makes use of both existing algorithms from past literatures and newly developed tests. The most updated version, sts-2.1.1, which was released in August 11, 2010, is used in our randomness tests. TABLE 3.2 summarizes the 15 NIST tests [2].

Test Name	Test Description
1 The Frequency (Monobit) Test	Tests proportion of zeros and ones for the whole sequence.
2 Frequency Test within a Block	Tests the proportions of ones within M-bit Block.
3 The Run Test	Tests the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits
4 Tests for the Longest-Run-of-Ones in a Block	Tests the longest run of ones within M-bit Block and consistency with theory
5 The Binary Matrix Rank Test	Tests rank of disjoint sub-matrices of the entire sequence and independence
6 The Discrete Fourier Transform (Spectral) Test	Tests the peak heights in the Discrete Fourier Transform of the sequence, to detect periodic features that indicates deviation of randomness
7 The Non-overlapping Template Matching Test	Tests the number of occurrences of a pre-specified target strings
8 The Overlapping Template Matching Test	Tests the number of occurrences of a pre-specified target strings. When window found, slide only one bit before the next search
9 Maurers Universal Statistics Test	Tests the number of bits between matching patterns
10 The Linear Complexity Test	Tests the length of a linear feedback shift register, test complexity
11 The Serial Test	Tests the frequency of all possible overlapping m-bit pattern
12 The Approximate Entropy Test	Tests the frequency of all possible overlapping m-bits pattern across the entire sequence
13 The Cumulative Sums (Cusums) Test	Tests maximal excursion from the random walk defined by the cumulative sum of adjusted (-1, +1) digits in the sequence
14 The Random Excursion Test	Tests the number of cycles having exactly K visits in a cumulative sum random walk
15 The Random Excursions Variant Test	Tests the total number of times that a particular state is visited in a cumulative sum random walk

Table 3.2: Summary of the NIST test suite

Figure 3.10 shows one test result for the even-odd scheme, which only used an LSB from the up-time and down-time, when bits with both RTN and thermal noise are used. 10 sequences generated from multiple bits are tested and each sequence consists of 600,000 bits. Note that some of the results are not shown here due to the space constraint. NonOverlappingTemplate, RandomExcursions and RandomExcursionsVariant have a lot of tests. In the result above, the proportion in the second column shows the proportion of the sequences which passed the test. If the proportion is greater than or equal to the threshold value specified at the bottom of the figure (8 out of 10 or 4 out of 5), then the data is considered random. The P-value in the first column indicates the uniformity of the P-values calculated in each test. If P-value is greater than or equal to 0.0001, the sequences can be considered to be uniformly distributed [2]. The result indicates that the proposed RNG passes all the NIST tests.

We also tested random numbers from one bit with only RTN behavior, using multiple bits from up-time and down-time. In this case, we generated ten 200,000-bit sequences from one bit. The data passed all NIST tests with results that are similar to the above case. For the Universal test, which requires a sequence longer than 387,840 bits, we used five 500,000-bit sequences.

3.2.3 Performance

The throughput of the proposed RNG varies significantly depending on the switching rate of individual bits, sampling speed and environment conditions. Typically, only a small fraction of bits show pure RTN behavior with minimal thermal noise. TABLE 3.3 shows the performance of Flash chips from four

```
------------------------------------------------------------------
RESULTS  FOR  THE  UNIFORMITY  OF  P-VALUES  AND  THE
PROPORTION OF PASSING SEQUENCES
------------------------------------------------------------------
   P-VALUE         PROPORTION         STATISTICAL TEST
   0.122325        10/10              Frequency
   0.911413        10/10              BlockFrequency
   0.534146        10/10              CumulativeSums
   0.066882        10/10              CumulativeSums
   0.534146        10/10              Runs
   0.350485        10/10              LongestRun
   0.739918        10/10              Rank
   0.739918        10/10              FFT
   0.213309        10/10              NonOverlappingTemplate
   0.739918        10/10              NonOverlappingTemplate
   0.350485        10/10              OverlappingTemplate
   0.911413         9/10[1]           Universal
   0.534146        10/10              ApproximateEntropy
     ----           5/5               RandomExcursions
     ----           5/5               RandomExcursions
     ----           5/5               RandomExcursionsVariant
     ----           5/5               RandomExcursionsVariant
   0.739918        10/10              Serial
   0.350485        10/10              Serial
   0.534146        10/10              LinearComplexity
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
[1]The minimum pass rate for each statistical test is 8 for a sample size of
10 binary sequences, and 4 for a sample size of 5 sequences.
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 3.10: NIST test suite results for bits with RTN and thermal noise.

manufacturers. The average throughput ranges from 848 bits/second to 3.37 Kbits/second. Note that the fastest switching trap that can be identified is limited by the reading speed in our experiments.

If bits with both RTN and thermal noise are also used, the percentage of bits which can be used for RNG can be much higher. The performance of these bits from the same Flash chips as in the pure RTN case is shown in TABLE 3.4. The average throughputs are higher because thermal noise is high frequency noise.

In our tests, the RNG throughput is largely limited by the timing of the asyn-

Chip	Hynix SLC	Numonyx SLC	Micron SLC	Micron MLC
Reading speed (KHz)	46.51	45.25	43.10	17.78
Number of bits characterized	303	478	1030	134
Number of bits identified	9	16	5	0
Max throughput (bits/sec)	8.03K	5.35K	2.71K	--
Ave. throughput (bits/sec)	3.27K	1.79K	848.29	--
Min throughput (bits/sec)	107.04	34.77	8.14	--

Table 3.3: Performance of bits with pure RTN behavior.

Chip	Hynix SLC	Numonyx SLC	Micron SLC	Micron MLC
Reading speed (KHz)	46.51	45.25	43.10	17.78
Number of bits characterized	303	478	1030	134
Number of bits identified	27	81	58	28
Max throughput (bits/sec)	11.48K	9.68K	10.03K	3.83K
Ave. throughput (bits/sec)	3.28K	3.87K	3.53K	1.26K
Min throughput (bits/sec)	28.39	10.21	8.14	55.12

Table 3.4: Performance of bits with both RTN and thermal noise.

chronous interface which is controlled by an ARM microcontroller with CPU frequency of 60MHz and the 8-bit bus for a Flash chip. We believe that the RNG performance can be much higher if data can be transferred more quickly through the interface. As an example, the average for RTN transition time is reported to range from 1 microsecond to 10 seconds [13]. If a 128 bytes can be read in 6 microseconds which is the ideal random cache read speed for the Micron SLC chips, a RTN bit with 0.1ms average transition time will give approximately 20 Kbits/second throughput. Note that one page could have multiple RTN bits and our algorithm allows using multiple bits in parallel so that the aggregated throughput of an RNG can be much higher. For example, if N bits can be read at a time, in theory, that can increase the throughput by a factor of N.

3.2.4 Temperature Variations

For traditional hardware RNGs, low temperatures present a particular challenge because thermal noise, which they typically rely on, can be reduced with the temperature. To study the effectiveness of the Flash-based RNG in low temperatures, we tested the scheme at two low temperature settings: one in a freezer, which is about -5C, and the other in dry ice, which is about -80C. The generated random sequences are tested individually as well as combined together with data from experiments at room temperature. All of them passed the NIST test suite without a problem, showing that our technique is effective at low temperatures.

Note that the experiments for temperature variations and aging are performed with a setup where data from Flash memory are transferred from a testbed to a PC through an USB interface. The post processing is performed on the PC. The USB interface limits the Flash read speed to 6.67KHz. As a result, the throughput in this setup is noticeably slower than the results in previous subsections where the entire RNG operation is performed on a microcontroller.

To understand the impact of temperature variations on the Flash-based RNG, we tested the first 80 bits of a page from a Numonyx chip. At room temperature, 62 bits out of the 80 bits showed oscillations between the programmed state and erased state. 14 bits out of the 62 bits were selected by the selection algorithm, which identifies bits with pure RTN or both RTN and thermal components. The throughputs of the 14 bits are shown in Figure 3.11.

Figure 3.12 and Figure 3.13 show the performance of the RNG at -5 C and -80 C, respectively. At -5 C, 79 bits out of 80 bits showed noisy behavior and 20
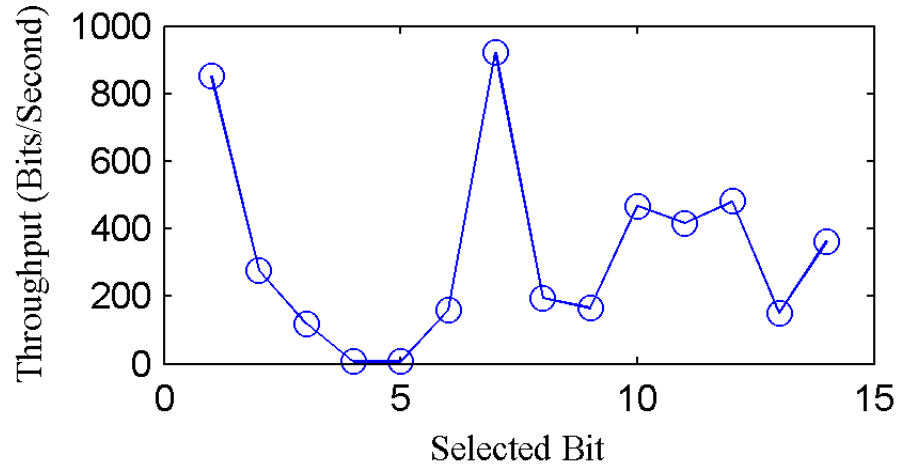
Figure 3.11: Throughputs under room temperature.

out of 79 bits were selected by the RNG algorithm as ones with RTN. At -80 C, 72 bits out of 80 bits showed noise and 28 out of 72 bits were selected as the ones with RTN. On average, we found that per-bit throughput is slightly decreased at low temperatures, most likely because of reduced thermal noise and possibly because of slowed RTN switching. However, the difference is not significant. In fact, a previous study [14] claimed that RTN is temperature independent below 10 Kelvin. Interestingly, we found that the number of bits that are selected by our algorithm as ones with RTN behavior increases at a low temperature. This trend is likely to be because the low temperature decreases thermal noise amplitude while RTN amplitude stays almost the same and the RTN traps slow down so that they become observable at our sampling frequency.

3.2.5   Aging

Flash devices wear-out over time as more program/erase (P/E) operations are performed. A typical SLC Flash chip has a lifetime of 1 million P/E cycles. In the
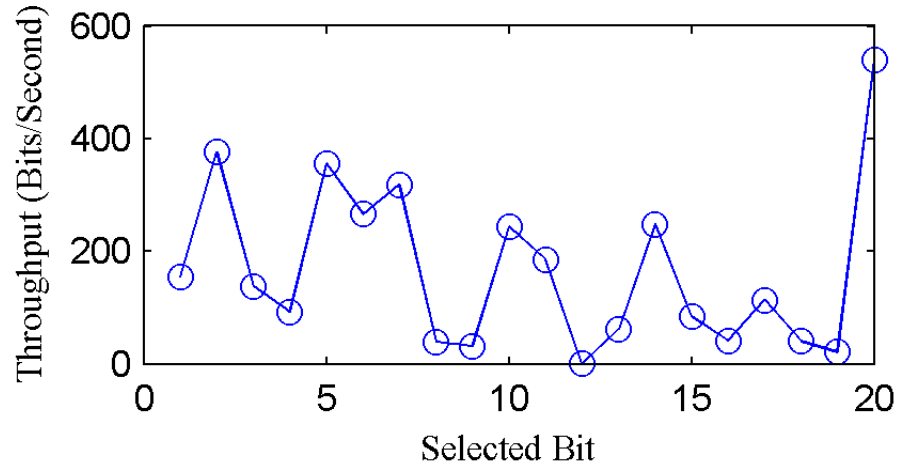
Figure 3.12: Throughput at -5 C.
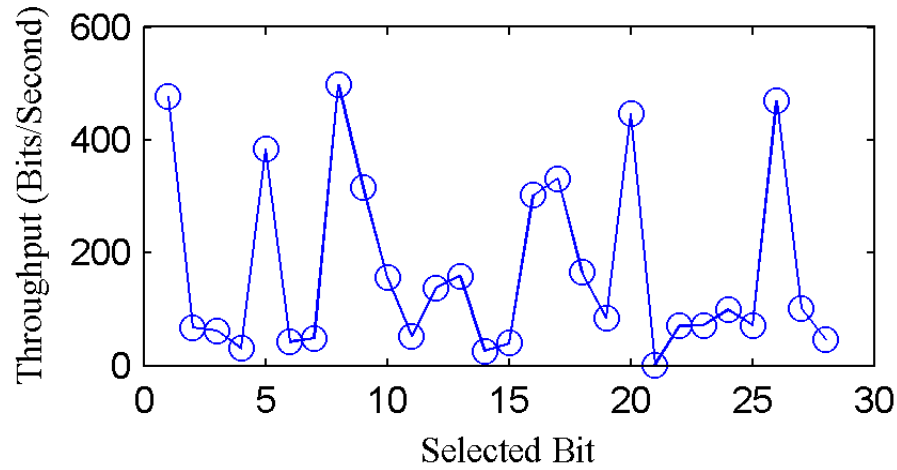
Figure 3.13: Throughputs at -80 C.

context of RNGs, however, we do not think that wear-outs cause concerns. In fact, aging can create new RTN traps and increase the number of bits with RTN. To check the impact of aging on the RNG, we tested the scheme after 1,000 P/E operations and 10,000 P/E operations as shown in TABLE 3.5. The RNG outputs passed the NIST test suite in both cases and did not show any degradation in performance.

Stress (P/E)	Bits with noise	Bits selected	Ave. throughput (bits/sec)
1,000	64	9	303.26
10,000	70	15	239.66

Table 3.5: Performance summary of RTN in stressed pages

The table shows an interesting trend that more bits show RTN behavior after 10,000 P/E cycles. The increase in noisy bits can potentially increase the overall RNG throughput. One possible concern with aging is a decrease in stable time period during which each bit shows noisy behavior. In our experiments, we found that a bit can be used for random number generation for over 12 hours after one programming (Algorithm III). If a bit is completely worn out, charge can leak out more quickly, requiring more frequent calibration. However, given that Flash memory is designed to have a retention time of 10 years within its lifetime, we do not expect the leakage to be a significant problem. We plan to perform larger scale experiments to understand how often a bit needs to be re-programmed for reliable random number generation. In practice, a check can also be added to ensure that a bit oscillates between 1 and 0.

3.3 Application Scenarios

This section briefly discusses how the Flash memory based security functions, namely RNGs and device fingerprints, can be used to improve security of electronic devices. We first discuss where the techniques can be deployed and present a few use cases.

The proposed Flash-based security techniques work with commercial off-the-shelf Flash memory chips using standard interfaces. For example, our pro-

totype design is based on the Open NAND Flash Interface (ONFI) [15], which is used by many major Flash vendors including Intel, Hynix, Micron, and SanDisk. Other Flash vendors such as Samsung and Toshiba also use similar interfaces to their chips. The proposed techniques can be applied to any Flash or other floating-gate non-volatile memory, as long as one can control read, program (write), and erase operations to specific memory locations (pages and blocks), issue the RESET command and disable internal ECC. Embedded systems typically implement a Flash memory controller in software, exposing the low-level Flash chip interface to a software layer. Our prototype USB board in the evaluation section is an example of such a design. While we did not have a chance to study details, the manual for the TI OMAP processor family [16], which is widely used in mobile phones, indicates that its External Memory Interface (EMI) requires software to control each phase of NAND Flash accesses. In such platforms where Flash accesses are controlled by software, our techniques can be implemented as relatively simple software changes.

For large memory components such as SSDs, the low-level interfaces to Flash memory chips may not be exposed to a system software layer. For example, SSD controllers often implement wear-leveling schemes that move data to a new location on writes. In such devices, the device vendor needs to either expose the Flash interfaces to higher level software or implement the security functions in firmware.

The Flash-based random number generator (RNG) can either replace or complement software pseudo random number generators in any applications that need sources of randomness. For example, random numbers may be used as nonces in communication protocols to prevent replays or used to generate new

33

cryptographic keys. Effectively, the Flash memory provides the benefits of hardware RNGs for systems without requiring custom RNG circuits. For example, with the proposed technique, low-cost embedded systems such as sensor network nodes can easily generate random numbers from Flash/EEPROM. Similarly, virtual machines on servers can obtain true random numbers even without hardware RNGs.

3.4   Related Work

Hardware random number generators generate random numbers from high-entropy sources in the physical world. Theoretically, some random physical processes are completely unpredictable. Therefore, hardware random number generators provide better random numbers in terms of randomness than software based pseudo-random number generators.

Thermal noise and other system level noise are the common entropy sources in recently proposed hardware random number generators. In [17], the phase noise of identical ring oscillators is used as the entropy source. In [18], the differences in path delays are used. In [19] and [20], the metastability of flip-flops or two cross coupled inverters are used. Basically, the entropy source of these RNG designs is thermal noise and circuit operational conditions. These hardware random number generators can usually achieve high throughput because the frequency of the entropy sources is high. One common characteristic of these hardware random generators is that they all need carefully designed circuits where process variations should be minimized so that noises from the entropy source can be dominant. Compared to this, the random number generation in

Flash memory cells does not require specially designed circuits and is more immune to process variation. Moreover, our entropy source is based on quantum behavior and theoretically, it should still work under extremely low temperatures where thermal noise or other kinds of noise decrease dramatically.

RANDOM NUMBER GENERATION

4.1 Theory and Implementation

This section describes techniques to generate unique fingerprints from Flash memory devices.

4.1.1 Sources of Uniqueness

Flash memory is subject to random process variation like any other semiconductor device. Because Flash is fabricated for maximum density, small variations can be significant. Process variation can cause each bit of a Flash memory to differ from its neighbors. While variation may affect many aspects of Flash cells, our fingerprinting technique exploits threshold voltage variations. Variations in doping, floating gate oxide thickness, and control-gate coupling ratio can cause the threshold voltage of each transistor to vary. Because of this threshold voltage variation, different Flash cells will need different times to be programmed.

4.1.2 Extracting Fingerprints

In this paper, we introduce a fingerprinting scheme based on partial programming. We repeatedly partially program a page on a Flash chip. After each partial program, some bits will have been programmed enough to flip their states from 1 to 0. For each bit in the page, we record the order in which the bit flipped.

Choose a partial programming time T (below the rated program time).

Nbits = number of bits in one page
Order = 1;
Initialize BitRank[Nbits] to 0.

do {
   Partially program a page for T;
   For all programmed bits **do**
      BitRank[programmed bit] = Order;
   End for
   Order = Order + 1;
} repeat until most (99%) bits in the page are programmed

Figure 4.1: Extract the order in which bits in a page reach the programmed state.

Pseudo-code is provided in Algorithm V. In our experiments, T is chosen to be 29.3us. A short partial program time provide a better resolution to distinguish different bits with the cost of increased fingerprinting time. We do not enforce all bits to be programmed, in order to account for the possibility of faulty bits.

4.1.3 Comparing Fingerprints

The fingerprints extracted from the same page on the same chip over time are noisy but highly correlated. To compare fingerprints extracted from the same page/chip and different pages/chips, we use the Pearson correlation coefficient [5], which is defined as

$$P(x, y) = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \tag{4.1}$$

where X is the vector of program orders extracted from one experiment and Y is another vector of program orders extracted from another experiment. μ_X and σ_X are the mean and standard deviation of the X vector. μ_Y and σ_Y are the mean and standard deviation of the Y vector.

In this way, the vector of program orders is treated as a vector of realizations of a random variable. For vectors extracted from the same page, $Y = aX + b + noise$ where a and b are constants and the noise is small. So, X and Y are highly correlated and the correlation coefficient should be close to 1. For vectors extracted from different pages, X and Y should be nearly independent of each other, so the correlation coefficient should be close to zero. From another perspective, if both X[i] and Y[i] are smaller or bigger than their means, $(X[i] - \mu_X)(Y[i] - \mu_Y)$ would be a positive number. If not, it would be a negative number. If X and Y are independent, it is equally likely to be positive and negative so the correlation coefficient would approach 0.

The scatter plot of X and Y from the same page/chip and from different chips are shown in Figure 4.2. The figure clearly demonstrates a high correlation between fingerprints from the same chip over time and a low correlation between fingerprints from different chips. Therefore, this correlation metric can be used to compare fingerprints to determine whether they are from the same page/chip or from different pages/chips.

4.1.4 Fingerprints in Binary Numbers

The above fingerprints are in the form of the order in which each bit was programmed. If an application requires a binary number such as in generating

correlation coefficient=0.9793

Number of Relative Programs Trial 2

Number of Relative Programs, Trial 1

(a)

correlation coefficient = -0.0039

Number of Programs, Chip 19

Number of Programs, Chip 22
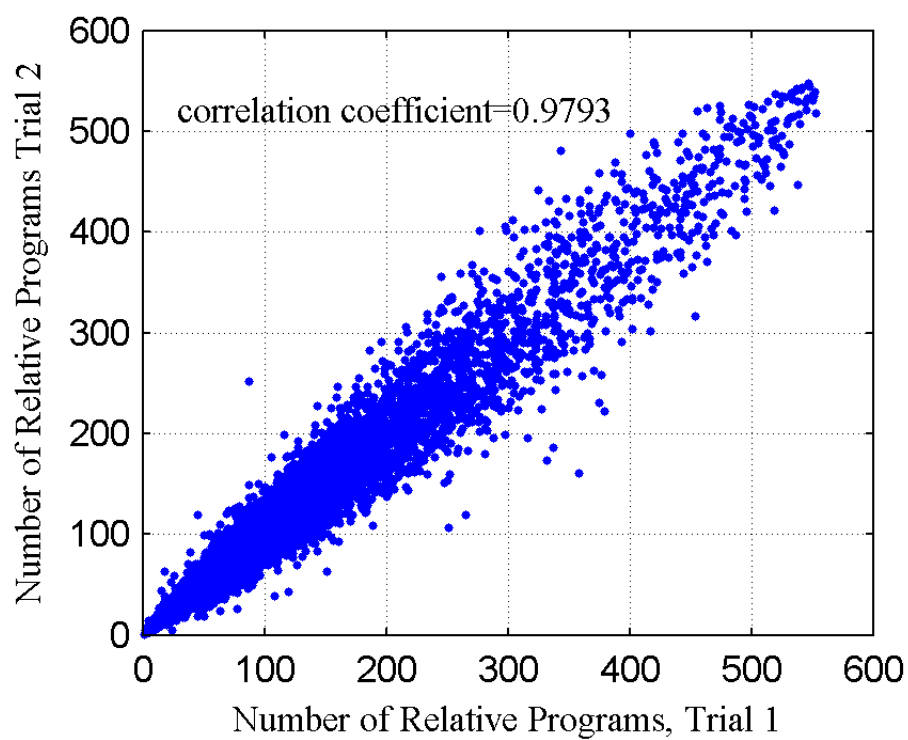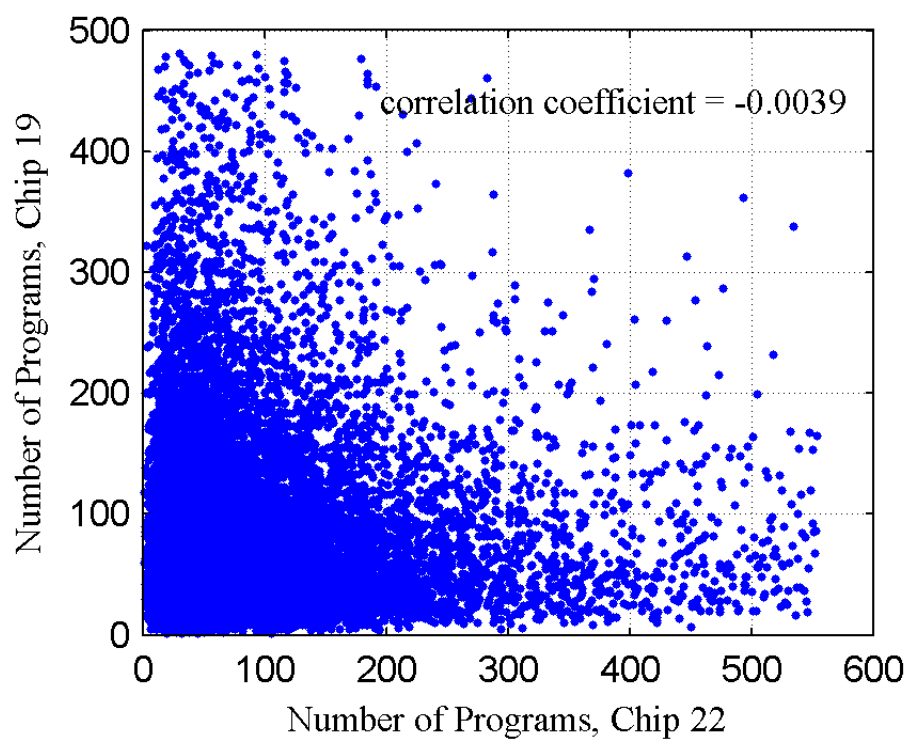
(b)

Figure 4.2: Scatter plot for fingerprints extracted on (a) the same page and (b) different chips.

39

> **Algorithm VI Generate a binary signature from the partial programming order information.**
>
> Pick threshold $t = Max(BitRank)/2$
> **For** each bit
>   **If** $BitRank[bit] > t$
>   Output 1
>   **Else** Output 0
> **End for**

Figure 4.3: Generate a binary signature from the partial programming order information.

cryptographic keys, we need to convert the recorded ordering into a binary number.

There are a couple of ways to generate unique and unpredictable binary numbers from the Flash fingerprints. First, we can use a threshold to convert a fingerprint based on the programming order into a binary number as shown in Algorithm VI. In the algorithm, we produce 1 if the program order is high, or 0 otherwise. This approach produces a 1 bit fingerprint for each Flash bit. Alternatively, we can obtain a similar binary fingerprint directly from Flash memory by partially programming (or erasing) a page and reading bits (1/0) from the Flash.

4.2 Evaluation

The experiment setup and tested devices are the same as in the previous chapter.

For fingerprinting, we are interested in uniqueness and robustness of fingerprints. The fingerprint should be unique, which means that fingerprints from different chips or different locations of the same chip must be significantly dif-

ferent the correlation coefficient should be low. The fingerprint should also be robust, in a sense that fingerprints from a given location of a chip must stay stable over time and even under different environmental conditions the correlation coefficient should be high.

In the experiments detailed below, we used 24 chips (Micron 34nm SLC), and 24 pages (6 pages in 4 blocks) from each chip. 10 measurements were made from each page. Each page has 16,384 bits.

4.2.1 Uniqueness

To test uniqueness, we compared the fingerprint of a page to the fingerprints of the same page on different chips, and recorded their correlation coefficients. A total of 66,240 pairs were compared (24 chips choose 2) * 24 pages * 10 measurements. The results are shown in Figure 4.4. The correlation coefficients are very low, with an average of 0.0076. A Gaussian distribution fits the data well, as shown in red.

The correlation coefficients are also very low when a page is compared not only to the same page on different chips, but also to different pages on the same and different chips, shown in Figure 4.5. There are 1,656,000 pairs in comparison ((24 pages * 24 chips) choose 2) * 10 measurements. This indicates that fingerprints from different parts (pages) of a chip can be considered as two different fingerprints and do not have much correlation. Therefore, the fingerprinting scheme allows the generation of many independent fingerprints from a single chip. The average correlation coefficient in this case is 0.0072.
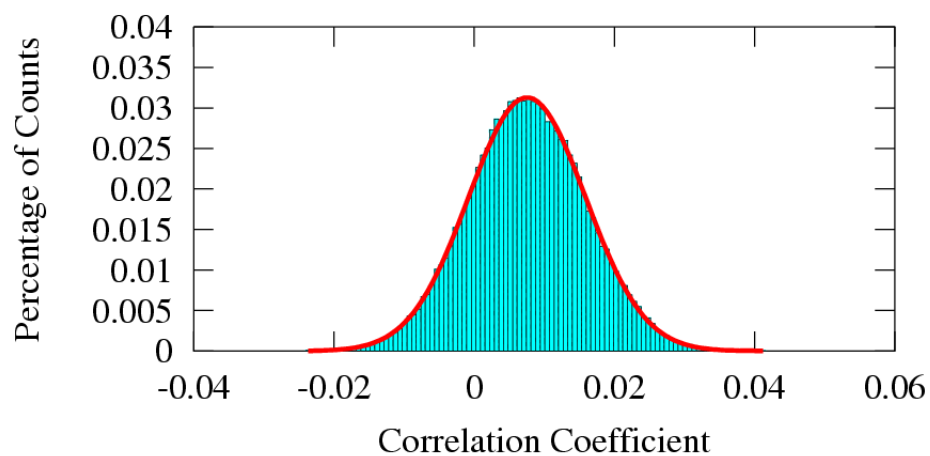
Figure 4.4: Histogram of correlation coefficients for pages compared to the same page on a different chip (total 66,240 comparisons).
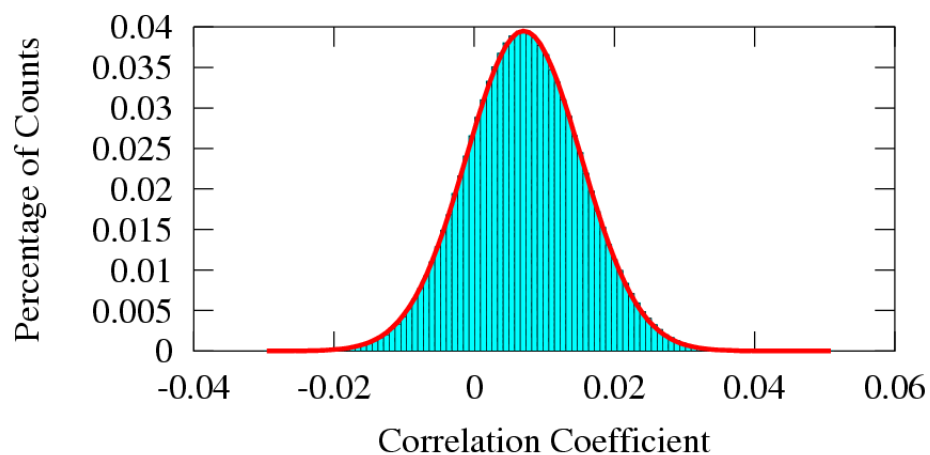
Figure 4.5: Histogram of correlation coefficients for every page compared to every other page at room temp (total 1,656,000 comparisons).
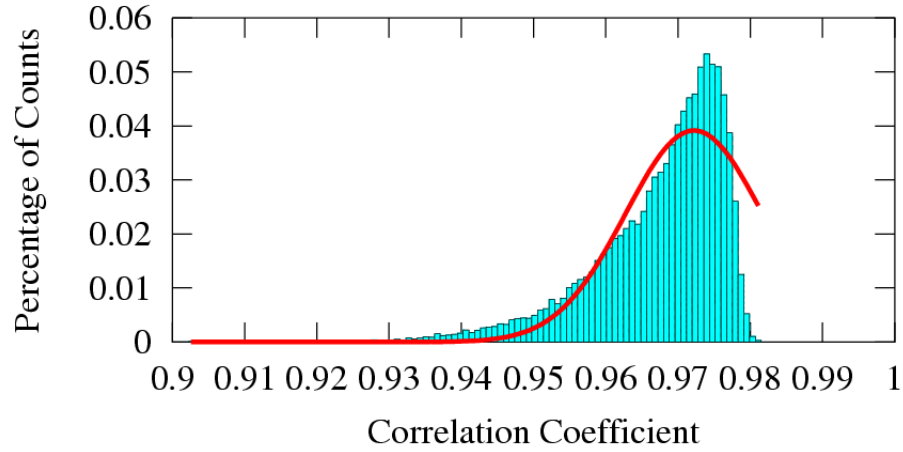
Figure 4.6: Histogram of correlation coefficients for all intra-chip comparisons (total 25,920 comparisons).

4.2.2 Robustness

To test robustness, we compared each pages measurement to the 9 other measurements of the same pages fingerprint (an intra-chip measurement). The histogram of results for all pages is shown in Figure 4.6. The correlation coefficient for fingerprints from the same page is very high, with an average of 0.9673. The minimum observed coefficient is 0.9022. The results show that fingerprints from the same page are robust over multiple measurements, and can be easily distinguished from fingerprints of a different chip or page.

To be used in an authentication scheme, one could set a threshold correlation coefficient t. If, when comparing two fingerprints, their correlation coefficient is above t, then the two fingerprints are considered to have come from the same page/chip. If their correlation coefficient is below t, then the fingerprints are assumed to be from different pages/chips.

In such a scheme, there is a potential concern for false positives and false

negatives. A false negative is defined as comparing fingerprints that are actually from two different pages/chips, but deciding that the fingerprints are from the same page/chip. A false positive occurs when comparing fingerprints from the same page/chip, yet deciding that the fingerprints came from two different pages/chips. The threshold t can be selected to balance false negatives and positives. A high value of t would minimize false negatives, but increase the chance of false positives, and vice versa.

To estimate the chance of false positives and false negatives, we fit normal probability mass distribution functions to the correlation coefficient distribution. A false positive would arise from a comparison of two fingerprints from the same page being below t. The normal distribution fitted to the intra-chip comparison data in Figure 4.6 has an average $\mu = 0.9722$ and a std. deviation of 0.0095. For a threshold of $t = 0.5$, the normal distribution function estimates the cumulative probability of a pair of fingerprints having a correlation coefficient below 0.5 as $2.62 * 10^{539}$. At t = 0.7, the probability is estimated as $7.43 * 10^{-181}$.

The normal distribution function fitted to the inter-chip comparison data in Figure 4.5 has a $\mu = 0.0076$ and a std. deviation of 0.0083. The estimated chance of a pair of fingerprints from different chips exceeding $t = 0.5$ is $4.52 * 10^{-815}$. At $t = 0.3$, the probability is estimated as $6.14 * 10^{-301}$.

The tight inter-chip and intra-chip correlations along with low probability estimates for false positives or negatives suggest that the size of fingerprints can possibly be reduced. Instead of using all 16,384 bits in a page, we can generate a fingerprint for a 1024-bit, 512-bit, or even only a 256-bit block. Experiments show that the averages of the observed correlation coefficients remain similar to those when using every bit in a page while the standard deviation increases

by a factor of 2-3. However, the worst-case false negative estimates remain low. When using 256 bit fingerprints with the threshold $t = 0.3$, the estimate is $7.91 * 10^{-7}$. Under the same conditions, using 1024 bit fingerprints gives an estimated $3.20 * 10^{-22}$ chance of a false negative.

4.2.3   Temperature Variations and Aging

To see how robust the fingerprints are across different temperatures. We extracted fingerprints from chips at two other ambient temperatures, 60 C and -5 C. We tested a subset of the chips tested at room temperature 6 pages (3 pages in 2 blocks) in 6 chips.

Of interest is how fingerprints from the same page/chip, but taken at different temperatures, compare. Figure 4.7 shows the results of the intra-chip comparison between each temperature pair. Correlations remain high for fingerprints from the same page/chip, indicating that fingerprints taken at different temperatures can still be identified as the same. The average correlation coefficient is lower than when compared without a temperature difference, but is still sufficiently high to have very low false positive rates.

Comparing fingerprints from the same page at the same temperature at -5 C or 60 C still yields high correlation coefficients, as expected. Comparisons of fingerprints from different pages/chips at different temperatures give very low correlation coefficients.

Flash chips have a limited lifetime, wearing out over many program/erase (P/E) cycles. For a pages fingerprint to be useful over time, fingerprints taken
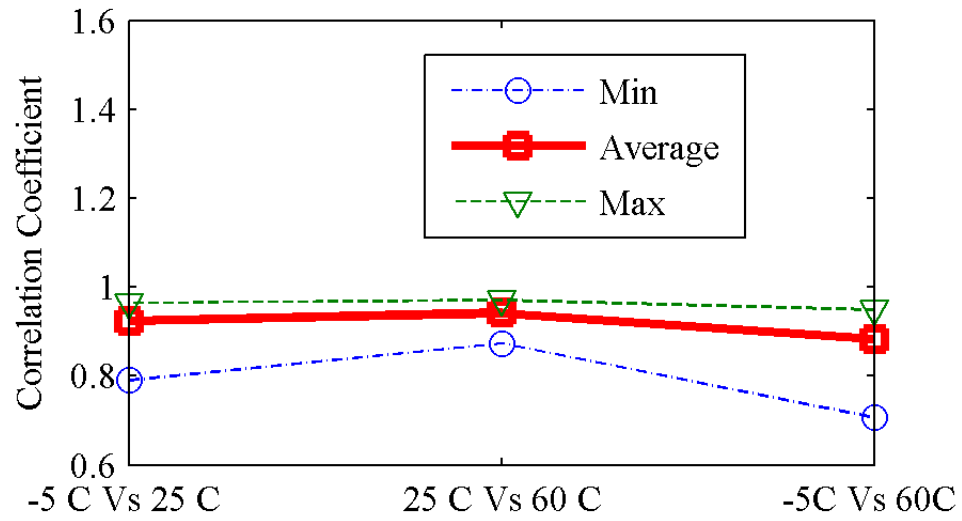
Figure 4.7: Average, minimum, and maximum correlation coefficients for intra-chip comparisons between different ambient temperatures.
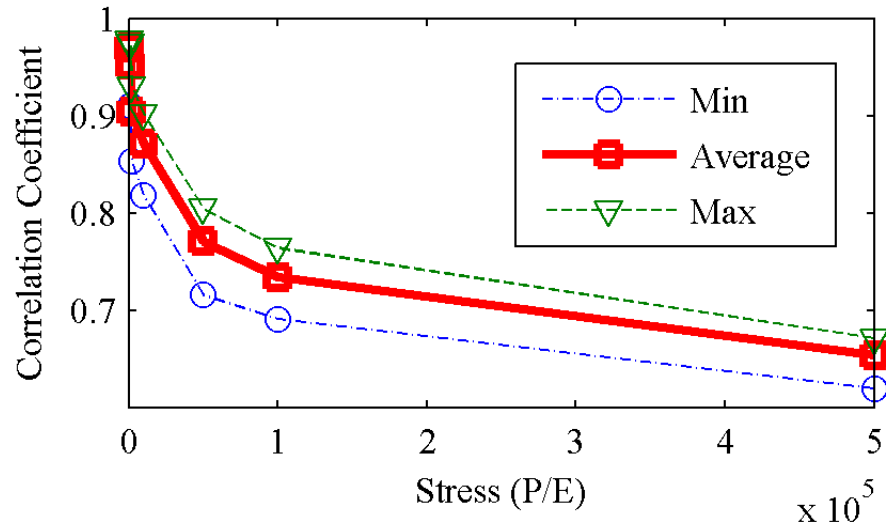
Figure 4.8: Average, minimum, and maximum correlation coefficients for comparisons between fresh and stressed Flash.

later in life should still give high correlation with younger fingerprints. Figure 4.8 shows the results of comparing fingerprints for the same page/chip taken when a Flash chip is new to fingerprints taken after a different number of P/E cycles. While the average correlation coefficient goes down noticeably, we note that it appears to bend towards an asymptote as the chip wears out. Even after 500,000 P/E cycles, which is beyond the typical lifetime of Flash chips, the average coefficient is still high enough to distinguish fingerprints of the same page/chip from fingerprints acquired from a different page/chip.

However, we found that an extreme wear-out such as 500,000 P/E cycles can raise a non-negligible false positive concern (10^{-4}) for short 256 or 512-bit fingerprints. This result indicates that we need longer fingerprints if they need to be used over a long period of time without a re-calibration.

4.2.4 Security

An attacker could attempt to store the fingerprints of a Flash device and replay the fingerprint to convince a verifier that he has the Flash chip in question. If the attacker cannot predict which page(s) or parts of a page (for shorter signatures) will be fingerprinted, he would need to store the fingerprints for every page to ensure success. The Flash chips in our experiments required about 800 partial program cycles per fingerprint. As the fingerprint comprises the order in which the bit was programmed, each bits ordering could be stored as a 10-bit number. To store an entire chips fingerprints would require 10x the chip storage.

Acquiring a single fingerprint is relatively fast. Our setup could record an entire pages fingerprint in about 10 seconds. However, there are 131,072 pages

on our (relatively small) test chip; characterizing one chip would take about 2 weeks. The characterization time depends on the speed of the Flash interface, and we plan to further investigate the limit on how fast fingerprints can be characterized.

4.2.5 Applicability to Multiple Flash Chips

Most of the above experimental results are obtained from the Micron SLC Flash memory. In order to answer the question of whether the proposed techniques are applicable to Flash memory in general, we have repeated both RNG and fingerprinting tests on four types of Flash memory chips in Table 5.1, including an MLC chip.

The experiments showed that RNG and fingerprinting both work on all four types of Flash chips, with comparable performance. Detailed results are not included as they do not add new information.

While we found that the proposed algorithm works without any change in most cases, there was one exception where the fingerprinting algorithm needed to be slightly modified in order to compensate for systematic variations for certain manufacturers. For example, for the Hynix and Numonyx chips, we found that bits from the even bytes of a page tend to be programmed quicker than bits from the odd bytes. Similarly, for the MLC chip, bits in a page divide into two groups: a quickly programmed group and a slowly programmed group. To accommodate such systematic behaviors, the fingerprinting algorithm was changed to only compare programming ordering of bits within the same group.

4.3   Application Scenarios

One application of the Flash device fingerprints is to identify and/or authenticate hardware devices themselves similar to the way that we use biometrics to identify humans.

As an example, let us consider distinguishing genuine Flash memory chips from counterfeits through an untrusted supply chain. Recent articles report multiple incidents of counterfeit Flash devices in practice, such as chips from low-end manufacturers, defective chips, and ones harvested from thrown-away electronics, etc. [5, 21, 22]. The counterfeit chips cause a serious concern for consumers in terms of reliability as well as security; counterfeits may contain malicious functions. Counterfeits also damage the brand name for a manufacturer.

The Flash fingerprints can enable authentication of genuine chips without any additional hardware modifications to todays Flash chips. In a simple protocol, a Flash manufacturer can put an identifier (ID) to a genuine chip (write to a location in Flash memory), generate a fingerprint from the chip, and store the fingerprint in a database along with the ID. To check the authenticity of a Flash chip from a supply chain, a customer can regenerate a fingerprint and query the manufacturers database to see if it matches the saved fingerprint.

In order to pass the check, a counterfeit chip needs to produce the same fingerprint as a genuine one. Interestingly, unlike simple identifiers and keys stored in memory, device fingerprints based on random manufacturing variations cannot be controlled even when a desired fingerprint is known. For example, even legitimate Flash manufacturers cannot precisely control individual transistor threshold voltages, which we use to generate fingerprints. To pro-
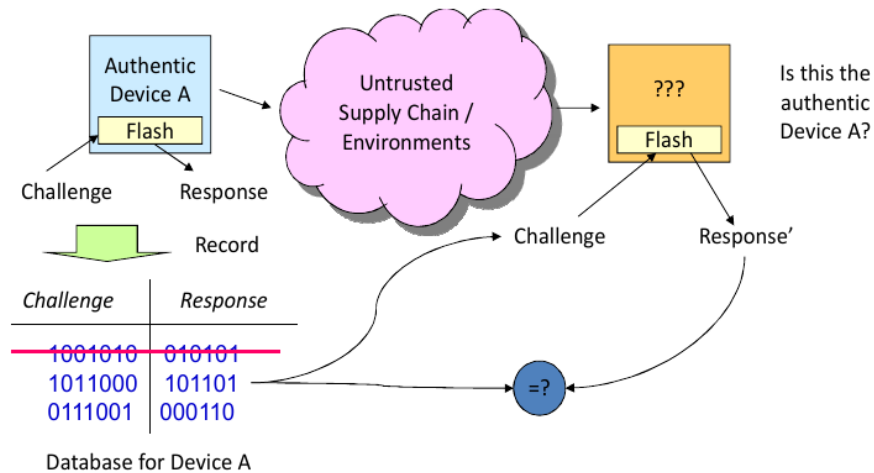
Figure 4.9: Device authentication through a challenge-response protocol.

duce specific fingerprints, one will need to create a custom chip that stores the fingerprints and emulates Flash responses.

The authentication scheme can be strengthened against emulation attacks by exploiting a large number of bits in Flash memory. Figure 4.9 illustrates a modified protocol that utilizes a large number of fingerprints that can be generated from each Flash chip. Here, we consider a Flash chip as a function where a different set of bits that are used to generate a fingerprint is a challenge, and the resulting fingerprint is a response. A device manufacturer, when in possession of a genuine IC, applies randomly chosen challenges to obtain responses. Then, these challenge-response pairs (CRP) are stored in a database for future authentication operations. To check the authenticity of an IC later, a CRP that has been previously recorded but has never been used for a check is selected from the database, and a re-generated response from a device can be checked.

Unless an adversary can predict which CRPs will be used for authentication, the adversary needs to measure all (or at least a large fraction) of possible fin-

gerprints from an authentic Flash chip and store them in an emulator. In our prototype board, a generation of all fingerprints from a single page (16K bits) takes about 10 seconds and requires 10 bits of storage for each Flash bit. For a 16Gbit (2 GB) Flash chip, which is a moderate size by todays standards, this implies that fully characterizing the chip will take hundreds of days and 20 GB storage. In the context of counterfeiting, such costs are likely to be high enough to make producing counterfeits economically unattractive.

The security of the authentication scheme based on Flash fingerprints can be further improved if an additional control can be added to the Flash interface. For example, imagine using a USB Flash memory as a two-factor authentication token by updating its firmware to have a challenge-response interface for Flash fingerprints. Given that authentication operations only need to be infrequent, the USB stick can be configured to only allow a query every few seconds. If a fingerprint is based on 1024 Flash bits, fully characterizing an 8 GB USB stick can take tens of years.

In addition to device identification and authentication, the Flash fingerprints can be used as a way to produce many independent secret keys without additional storage. In effect, the proposed Flash fingerprints provide unpredictable and persistent numbers for each device. Previous studies such as fuzzy extractors [23] and Physical Unclonable Functions (PUFs) [3] have shown how symmetric keys (uniformly distributed random numbers) can be obtained from biometric data or IC signatures from manufacturing variations by applying hashing and error correction. The same approach can be applied to Flash fingerprints in order to generate reliable cryptographic keys. A typical Flash with a few GB can potentially produce tens of millions of 128-bit symmetric keys.

4.4 Related Work

Instead of conventional authentication based on a secret key and cryptographic computation, researchers have recently proposed to use the inherent variation in physical characteristics of a hardware device for identification and authentication. Process variation in semiconductor foundries is a common source of hardware uniqueness which is out of the control of the designer [24, 25, 26]. A unique fingerprint can be extracted and used to identify the chip, but cannot be used for security applications because it can be simply stored and replayed. We also take advantage of process variation for our fingerprinting scheme. For security applications, Physical Unclonable Functions (PUFs) have been proposed. A PUF can generate many fingerprints per device by using complex physical systems whose analog characteristics cannot be perfectly replicated. Pappu initially proposed PUFs [27] using light scattering patterns of optically transparent tokens. In silicon, researchers have constructed circuits which, due to random process variation, emit unique outputs per device. Some silicon PUFs use ring oscillators [28] or race conditions between two identical delay paths [29]. These PUFs are usually implemented as custom circuits on the chip. Recently, PUFs have been implemented without additional circuitry by exploiting metastable elements such as SRAM cells, which have unique value on start-up for each IC instance [4, 30], or in Flash memories [5]. Our authentication scheme requires no new circuitry and can be done with commercially available and ubiquitous Flash chips. Unlike metastable elements, authentication does not require a power cycle. The scheme can generate many fingerprints by using more pages in the Flash chip. Acquiring a fingerprint is also faster and more widely applicable than previous Flash authentication methods.

HIDING INFORMATION IN FLASH MEMORY

5.1 Overview

5.1.1 Threat Model

Figure 5.1 shows the overview of the information hiding process in Flash memory. In order to hide information in Flash, Alice (left) first adds an error correcting code (ECC) to her message payload and hides the payload in the analog characteristics in Flash memory. Later, Alice (right) can perform the reverse operations to retrieve the hidden payload by recovering bits from the analog characteristics and correct errors using the ECC. The information hiding and recovery algorithms use a secret key (hiding key) to determine where the hidden bits are stored in Flash memory. As error correcting codes are well studied, this paper focuses on the physical encoding and decoding of information in Flash.

As shown in the figure, an adversary (Eve) gets temporary access to the Flash memory after Alice hides information. We assume that the adversary can inspect and manipulate the memory through its normal interface, but do not consider physical tampering of the memory. In the simple case, the adversary can check normal Flash operations such as program, erase, and read operations. The adversary may also be aware of the information hiding technique and can specifically check analog characteristics of Flash memory that can be observed through the standard interface.

The goal of the adversary may differ depending on the target application. In
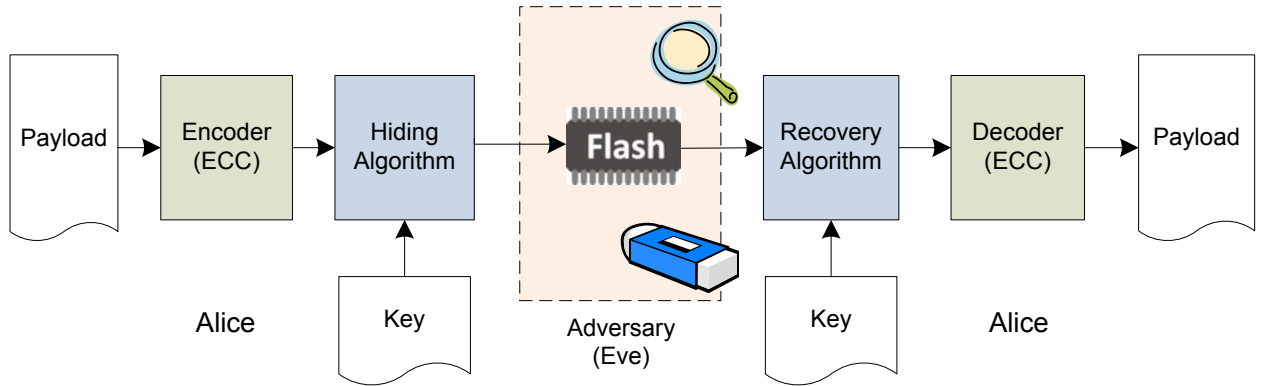
Figure 5.1: The overview of the information hiding operation.

particular, the adversary may try to

- Detect the existence of hidden information,

- Retrieve the hidden information, or

- Remove the hidden information.

For example, in the traditional steganography context where Alice is trying to establish a covert communication channel, it is important that the adversary cannot easily detect the existence of hidden information. On the other hand, in the context of storing sensitive information, it is more important that the adversary cannot retrieve information without knowing the hiding key. For watermarking, it should be difficult to erase the hidden information.

Given an unlimited amount of time with the Flash chip, an adversary can break the information hiding scheme by trying the retrieval algorithm on all pages with all possible hiding key values because we assume that an adversary knows our hiding algorithm. Therefore, the goal of the hiding technique is to

make the detection, retrieval, and removal of hidden information sufficiently time consuming for an attacker.

5.1.2   Flash Interface Requirements

The proposed technique is designed to work with Flash or other floating-gate non-volatile memory, as long as one can control read, program (write), and erase operations to specific memory locations (pages and blocks), issue the RESET command, and disable internal ECC (if there is any). For example, our experiments use off-the-shelf Flash chips that use the Open NAND Flash Interface (ONFI) [15], which is used by many major Flash vendors including Intel, Hynix, Micron, and SanDisk. Other Flash vendors such as Samsung and Toshiba also use similar interfaces to their chips. In many embedded and mobile devices, the required interface functions are already exposed to the software layers so that the proposed technique can be simply implemented as a software update.

5.2   Information Hiding Algorithm

This section describes the encoding (hiding) and decoding (recovery) algorithms for our information hiding scheme and the rationale for them.

5.2.1   Overview

Our scheme hides information in the program time of individual bits of Flash. The program time is the time it takes for a bit to change from the erased state
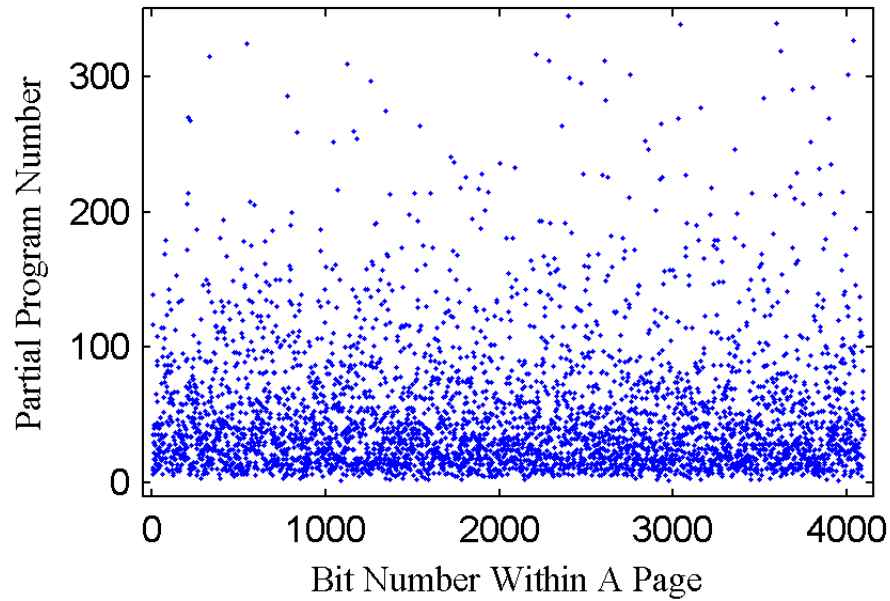
Figure 5.2: Raw partial program number for each bit in an example page.

(1) to the programmed state (0). Normally, a Flash memory controller performs a program operation at a page granularity, and the latency of this program operation is determined by the slowest bit in a page to be successfully written. In order to determine the program time for each bit, which we refer to as *per-bit program time*, we use the partial programming technique that is described in the previous section.

Figure 5.2 shows per-bit program times for a page. The plot shows the number of partial program operations to flip state from 1 to 0 for each bit in a page. Because of process variations, the program time varies widely from bit to bit as shown in the figure. The per-bit program time distribution for the page is shown in Figure 5.3. The wide distribution and noisy appearance of per-bit program times suggest that small changes to each bit's program time would go unnoticed, and could be used to carry a covert payload.
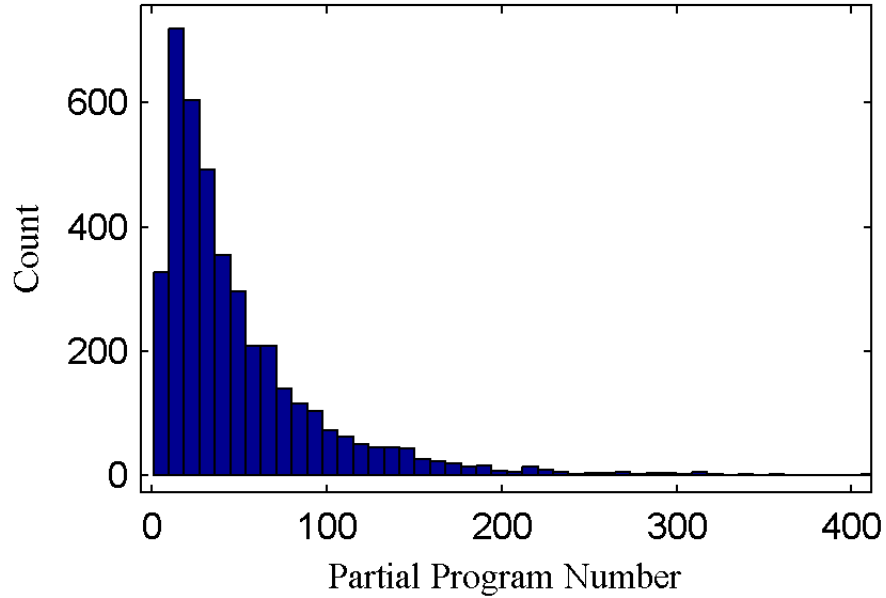
Figure 5.3: Partial program time distribution for bits in a page.

However, in order to hide information using the program time, we need to be able to intentionally change and control each bit's program time. Interestingly, in this context, previous work has observed that program time tends to decrease as a Flash cell becomes more worn-out [31, 5]. In this work, we also found that how worn-out each bit is can be controlled by selectively stressing a bit. Although one can only program an entire page together, we can stress some bits within a page more than others by controlling the value that we write. During an erase operation, every bit in a page is reset to an erased state (for example, assume that the erased state represents '1'). On a program operation, only bits that switch to 0 experience the program stress. When these bits are later erased, they also experience erase stress as they are reverted to the 1 state. Therefore, bits that undergo both switches (1 to 0 and 0 to 1) see the full program and erase stress from one program and erase cycle. However, bits that store 1 will not be switched to the 0 state by a program operation. These bits see much less

program and erase stress than their counterparts which are programmed to 0 because their states do not need to change. Therefore, by deciding whether to write a 1 or a 0 to each bit location in a page, we can control which bits are stressed more relative to other bits in the same page.

In theory, if every bit had a similar program time without much variation, we could hide one bit of information in every Flash bit by simply stressing or not stressing the bit so that its program time encodes the hidden bit. However, in practice, the program times of individual bits vary significantly due to manufacturing variations, and intentional stress is often not sufficient to overcome the inherent variations; inherently slow bits will be likely to be still slower than inherently fast bits even after being deliberately stressed. To address this issue, we choose to encode 1 bit of hidden information using many bits in Flash memory. For each bit to hide, we choose a group of Flash bits and program them to the same value, either 1 or 0. Effectively, this process encodes a bit in the collective program time of the group. The averaging effect reduces variations among different groups and allows the hidden bit to be more reliably recovered.

The use of a group also improves the security of the hiding scheme. In our scheme, we use a key (hiding key) to select which Flash bits will be grouped together for each hidden bit. If an attacker does not know the correct key, he or she cannot accurately identify which bits form a group together. Because an incorrect group is likely to contain both more stressed and less stressed bits, the average program time of an incorrect group of bits will not show a clear bias towards either 1 or 0.

For example, Figure 5.4 shows the distribution of the average program time of a correct group. In the experiment, we randomly selected 5,120 groups, each
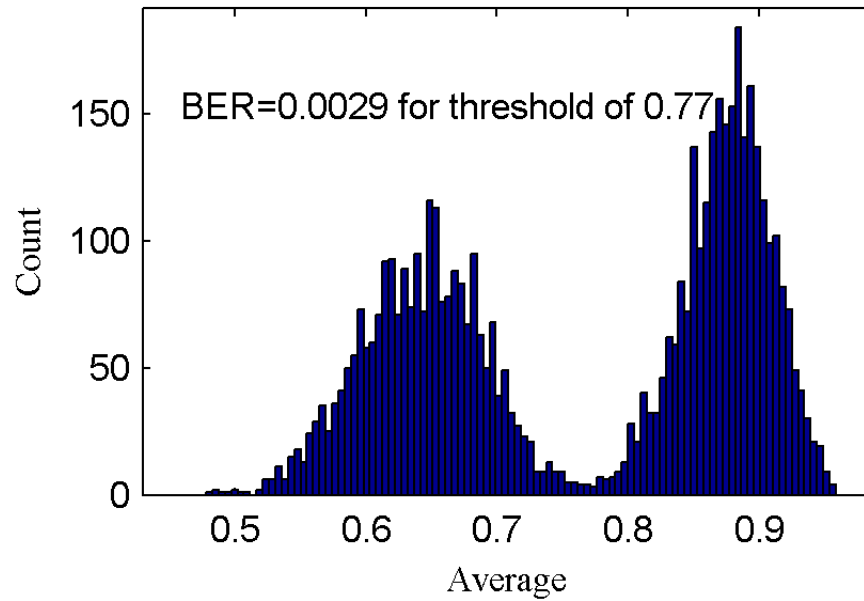
Figure 5.4: The distribution of the average program time of a group with
a correct key.

of which has 128 bits from a page, and hid either 1 or 0. As shown in the figure,
these is an obvious gap in the distribution between the fast and slow groups.
Therefore, the value of hidden bits can be easily recovered through a simple
thresholding.

On the other hand, Figure 5.5 shows the distribution of the average program
time when the hiding key is unknown. In this experiment, we used a randomly
selected hiding key. As shown in the figure, the average program time of a
group shows a normal distribution without any clear separation. This result
suggests that it is difficult for an adversary to recover hidden information with-
out correct groupings because each group is likely to have both more and less
stressed bits.
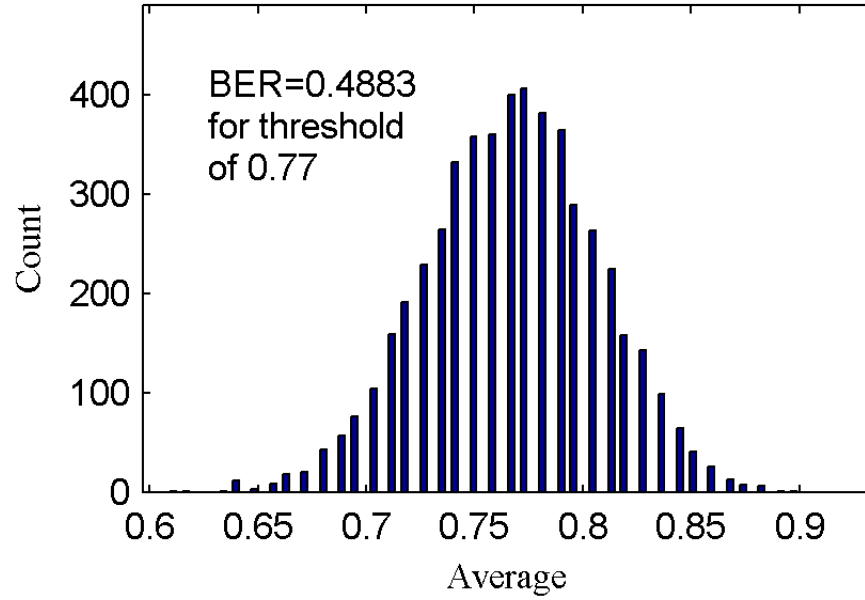
Figure 5.5: The distribution of the average program time of a incorrect group.

5.2.2 Hiding Algorithm

Figure 5.6 describes our methodology for hiding a payload in program time of Flash memory. The algorithm is split into two parts: (A) composing the payload by assigning bits of the message to groups of bits in Flash, and then (B) the actual process of writing the payload to Flash by repeated program and erase stress.

For a given message, we first choose a set of pages and blocks in which to encode the message based on the hiding key and the number bits that need to be hidden. Then, we divide the bits within each page into fixed size groups. Each group is used to store one message bit. The page, block, and group selections are based on the hiding key in a way that cannot be predicted without the key. In our implementation, we used RC4 to choose the Flash bit locations for each message bit.

Algorithm I: Encoding

Part A – Composing the message
1 For each selected page in a block
2   Generate the group for each message bit via the page hiding key
3   Assign each group 0 or 1 according to the embedded data
4   For each bit
5   If its group will represent a message "1"
6   Set it to be programmed 0
7   Else
8   Set it to be programmed 1
9   End if
10   End for
11 End for

Part B – Writing the message to Flash
1 For each selected block
2   For $i = 1, 2, .., N$ (N is the number of Hiding PE cycles)
3   Erase the block
4   Program every selected page
5   End for
6 End for

Figure 5.6: An algorithm to encode (hide) a payload into Flash memory program time.

Then, the algorithm determines which value (0 or 1) needs to be written to each bit location based on the message bit to be encoded. If a group is to store a "1" value, we will program (write a 0) the bits in the group, and the group will experience full program and erase stresses. If a group is to store a "0" value, the bits in the group will be set to 1, and will see less stress.

With the payload mapped to bits in Flash memory, we perform the actual write (program/erase) to Flash (Part B). We decide on a set number of stresses N to exert on the Flash. N is chosen to ensure an acceptable bit error rate without causing excessive stress. Each page is programmed N times in order to imprint the payload into the Flash. In our experiments, we found that several

hundred to a few thousand PE cycles are sufficient for SLC chips. An even smaller amount of PE cycles are enough for MLC chips.

5.2.3 Recovery Algorithm

Figure 5.7 describes our algorithm to decode a payload hidden by our encoding algorithm in Flash bit program time. Again, the algorithm is divided into two parts: (A) physically reading the per-bit program time from Flash, and (B) recomposing the payload from the program time distribution.

To read the hidden information, we must measure the program times for every bit in the pages containing the hidden bits. To do so, we use the partial programming algorithm described in the previous section. We choose M such that at the end of M partial programs, more than half of the bits, are programmed. The program time of a bit is expressed as the number of partial program cycles needed to flip the bit from 1 to 0. For the bits that do not flip after the M partial program operations, their program times are set to be a constant above M (i.e. $M + 1$).

To reconstruct the payload from the per-bit program times, we apply two thresholding steps. First, we compute the median program time X across all bits within each page. Then, the program time of each bit within a page is quantized based on the median; if a bit's program time is above half the median program time ($X/2$), then its program time is set to 1; otherwise it is set to 0. ($X/2$) was chosen empirically.

The bits are then divided into the groups specified by the hiding key. Within

Algorithm II: Decoding

Part A – Reading the program time from Flash
1 For each selected block
2   Erase the block
3   Program every bit in the block to 0
4   Erase the block
5   For each selected page
6   For $i = 1, 2, ..., M$
7   Partial program the page to 0 (abort a program operation after time T)
8   Read the page
9   For each bit in the page
10   If the bit changed from 1 to 0
11   Set $programtime$ for this bit to i
12   End if
13   End for
14   End for
15   For each bit
16   If the bit did not flip
17   Set its $programtime$ to be $M + 1$
18   End if
19   End for
20 End for
21 Erase the block
22 End for

Part B – Extracting the payload message
1 For each selected block
2   For each selected page
3   Calculate the median X of the program times for all the bits
4   For each bit
5   If its $programtime > (X/2)$
6   Set $programtime$ to 1
7   Else
8   Set $programtime$ to 0
9   End if
10   End for
11   Generate the group for each message bit with the page hiding key
12   For each group
13   Calculate the average program time for the group
14   If the average is less than Th
15   Recover the message bit: 1
16   Else
17   Recover the message bit: 0
18   End if
19   End for
20 End for
21 End for

Figure 5.7: An algorithm to decode (recover) a payload from Flash memory program time.

each group, the average of each individual bit's program times (now consisting of only 1 and 0) is computed, and the second thresholding step is performed. Each bit in the payload is set to 1 if the average program time of the corresponding group is below the threshold Th. Otherwise, the bit is set to 0.

In practice, with sufficient hiding PE cycles, we saw that there exists an obvious gap between the average program times of the more-stressed and less-stressed groups. As a result, it is straightforward to set the threshold Th to distinguish the two types of groups. For each page, we first sort the average program time of each group. Suppose the sequence of sorted program times is $X0, X1, X2, ..., XN$. Then we calculate the intervals between the sorted average program times and get $X1 - X0, X2 - X1, ...$ Suppose the maximum interval is $XM - XL$, then we set the threshold to be in the middle of that interval; $Th = (XM + XL)/2$. In this way, we can get a per-page threshold. For the cases with low hiding PE cycles, where there is no clear gap between the two clusters, the threshold is set to be a constant across pages based on the histogram of the average program times from multiple blocks.

For simplicity, we describe and evaluate the algorithm for the case where all bits within a selected page are used to hide bits. In order to make detection more difficult, it is also possible to only use a small subset of bits within a page. We leave this variant for future work.

5.3   Evaluation

In this section we evaluate the proposed scheme through experiments on Flash chips. In addition to validating correct operation of the encoding and decoding
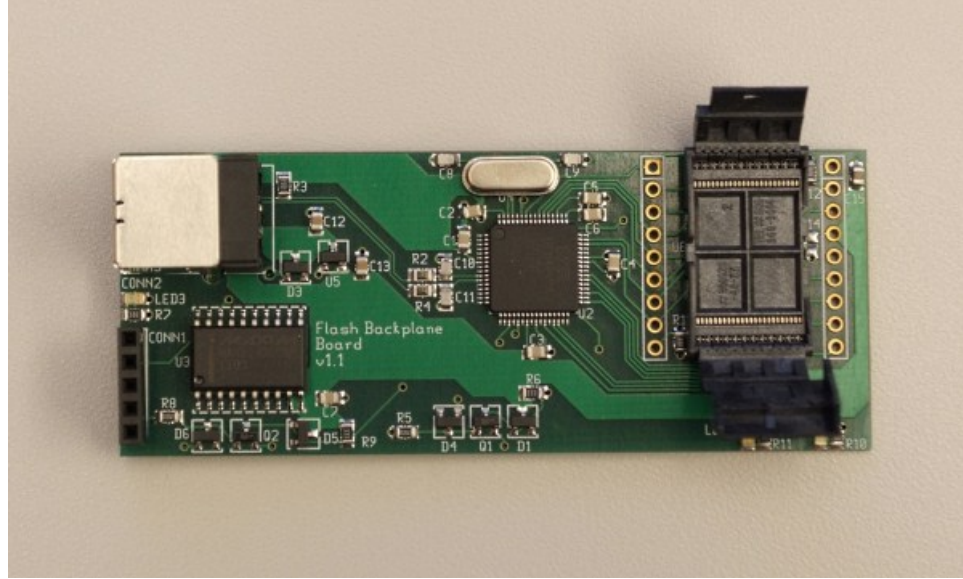
Figure 5.8: Flash test board.

algorithms, we also study the robustness across various design parameters, performance, detectability, recovery without the hiding key, and erase tolerance.

5.3.1 Evaluation Setup

Testbed Device

Our experiments use a custom Flash test board as shown in Figure 5.8. The board is made entirely with commercial off-the-shelf (COTS) components with a custom PCB. There is a socket to hold a Flash chip under test, an ARM microprocessor to issue commands and receive data from the Flash chip, and a Maxim MAX-3233 chip to provide a serial (RS-232) interface. USB support is integrated into the ARM microcontroller. We also wrote the code to test the device. The setup represents typical small embedded platforms such as USB Flash

Manufacturer	Part Number	Size	Qty	Process
Hynix	HY27UF084G2B	4 Gbit	1	5xnm class SLC
Micron	MT29F2G08ABA EAWP-IT:E4	2 Gbit	5	34nm SLC
Micron	MT29F4G08ABA DAWP:D	4 Gbit	15	34nm SLC
Micron	MT29F16G08CB ACAWP:C	16 Gbit	1	– MLC
Numonyx	NAND04GW 3B2DN6	4 Gbit	1	57nm SLC

Table 5.1: Tested Flash chips.

drives, sensor nodes, etc. This device shows that the techniques can be applied to commercial off-the-shelf devices with no custom integrated circuits (ICs).

Flash Memory Chips

The experiments in this paper were performed with five types of Flash memory chips from Numonyx, Micron, and Hynix. Table 5.1 shows their details. We primarily performed experiments with Micron 4Gbit chips. Experiments using other models will be marked.

In most experiments, we only used the first 4,096 bits of 16,896-bit pages to avoid performance overheads given the limited amount of memory in the microcontroller. We will refer to the first 4,096 bits as a "page" in the following discussion. For the analyses of per-page read/program time and per-block erase time, we used the entire page.
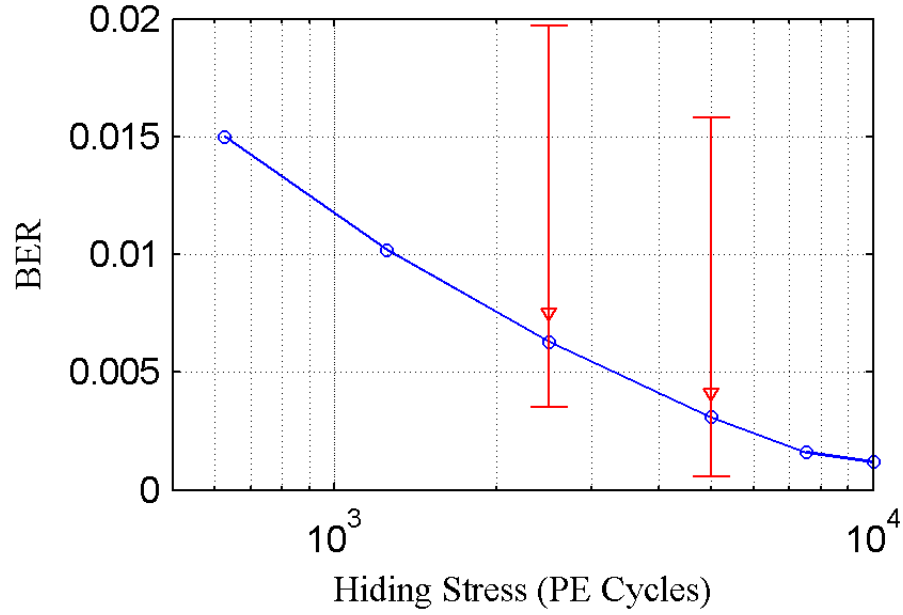
Figure 5.9: Influence of hiding stress on BER.

5.3.2 Robustness - Bit Error Rate

In this subsection, we first study whether the proposed scheme can reliably hide and recover bits in the program time characteristics. Here, we use the bit error rate (BER) as the metric for measuring robustness. To measure the BER, we hid a randomly generated message into Flash memory and compared the retrieved message with the original.

In the baseline experiment, we used the first 4,096 bits of a page and divided them into 32 groups (128 bits each) based on a randomly selected hiding key. Then, we selected multiple pages and blocks across a Flash chip to form 5,120 groups, which represent 5,120 hidden bits, and stored bits using 5,000 program and erase (PE) cycles in the encoding process. In this case, we got a bit error rate (BER) of 0.0029 (0.29%).
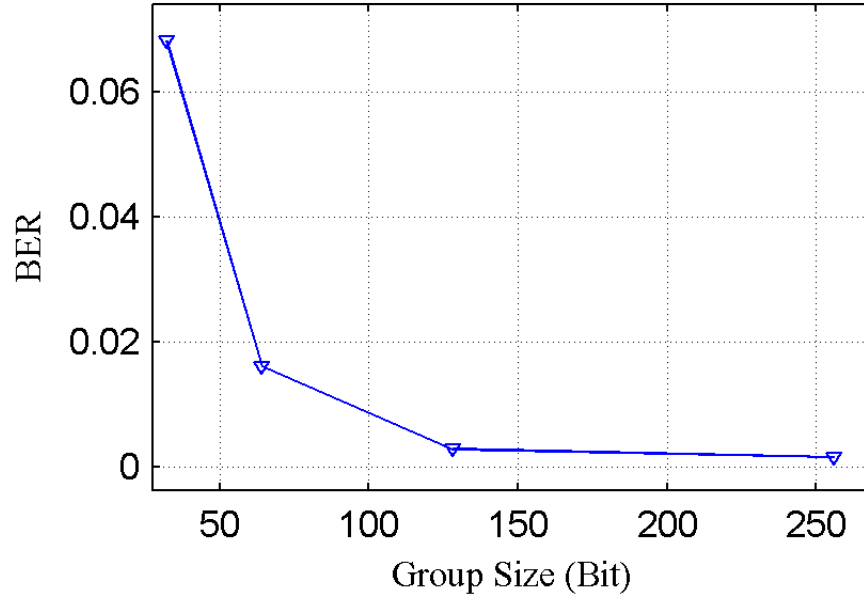
Figure 5.10: Influence of group size on BER.

Figure 5.9 shows the BER as a function of hiding stress, which is the number of program/erase (PE) cycles used to stress each group in the hiding process. The blue line shows the average BER using a single Micron 4Gbit chip. For each data point in the figure, the BER is computed over 5,120 bits of hidden information with the group size of 128 bits. For hiding stress levels of 2,500 and 5,000 PE cycles, we also show the statistics across 15 Flash chips; the red triangles show the average BER and the error bars show the maximum and minimum BERs across the 15 chips. We can see that the BER decreases as the hiding stress increases. More stress increases the program time difference between bits hiding 1s and 0s. However, the incremental benefit after 5,000 PE cycles is rather small. Note that the typical lifetime of an SLC Flash chip from the datasheet is 100,000 PE cycles.

There is also a trade-off between the robustness of the scheme and its hid-
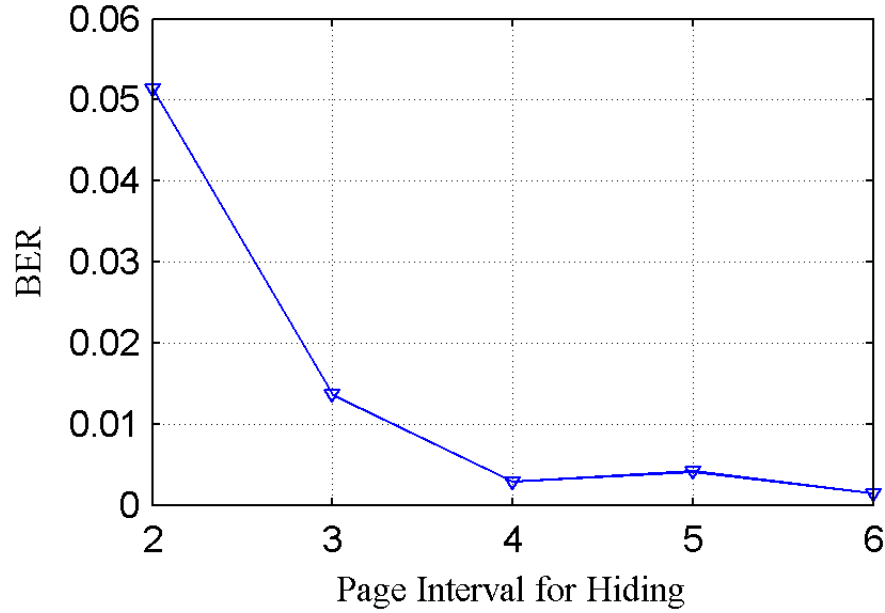
Figure 5.11: Influence of page interval on BER.

ing capacity. When more physical bits are included in a group, the capacity decreases. On the other hand, the statistical variations among groups will decrease as the group size increases. Therefore, the BER decreases with an increasing group size, as shown in Figure 5.10. It is also observed that neighboring pages have a strong influence on each other; stressing one page may also cause some stress in a neighboring page. To solve this problem, only a subset of pages with a specific interval K can be used within a block. If K is 4, then only page 0, page 4, page 8, and so on are used to hide information while the rest is not used. The influence of this page interval on the BER is shown in Figure 5.11. The experimental results suggest that there is not much benefit to using a group size beyond 128 and a page interval beyond 4 for these chips. Figure 5.10 and Figure 5.11 were generated from the 2Gbit Micron chips, but we found that the group size of 128 and page interval of 4 also work well for the 4 Gbit chips.
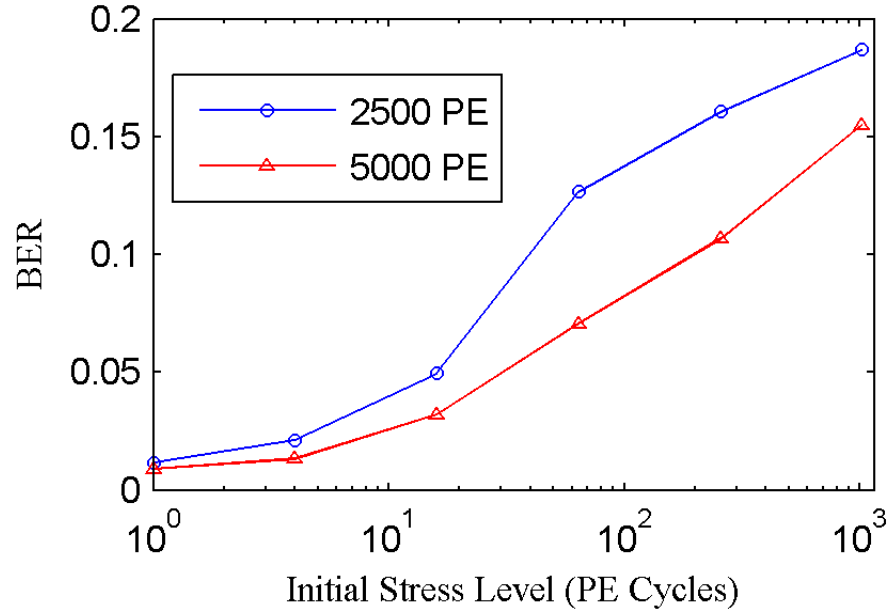
Figure 5.12: Influence of initial stress level on BER.

The effectiveness of the method on moderately used Flash chips is also studied. The influence of the initial stress level before the encoding process on the BER is shown in Figure 5.12. Here, we aim to simulate the normal usage of the Flash chip. So, in each program operation for the initial stress, random data are programmed. For example, the BER at the initial stress level of 10 PE cycles shows the error rate when bits are hidden after 10 PE cycles of programming random data. It can be observed that as the initial stress level increases, the BER also increases. However, a higher initial stress level can be tolerated by increasing the stress level in the encoding process. Note that the error rate is still manageable (less than 10-15%) even after hundreds of normal PE cycles.

The retention characteristics of the hiding scheme are shown in Table 5.2. Note that since each decoding performs 2 PE cycles, these retention characteristics include impacts from additional PE cycles in addition to the time between

	5,000 Hiding PE	10,000 Hiding PE
BER after zero retention (1 post PE cycle)	0.0029	0.0021
BER after 2-day retention (3 post PE cycles)	0.0141	0.0035
BER after 3-day retention (5 post PE cycles)	0.0187	0.0045
BER after over a month retention(7 post PE cycles)	0.0178	0.0031

Table 5.2: Retention characteristics of the hidden message.

information hiding and retrieval. In the first three rows of Table 5.2, the BER increases as retention time and post-hiding PE cycles increase. In the last row, the BER actually decreases a little compared to the third row. The results suggest that the retention time has little effect on the BER. Intuitively, given that the hiding scheme utilizes cell aging, this result is also supported by the fact that a worn-out Flash memory does not recover greatly even after having been left unattended for a long time.

5.3.3 Performance

In our experiments, when a whole page is used for hiding, it takes about 123.6 seconds to perform 5,000 PE cycles of hiding stress on a block, which embeds 2,048 bits of information in the block. The hiding throughput is around 16.6 bits/second. The upper limit of the throughput can also be calculated using the page program time and block erase time given in the Flash memory chip datasheet. The typical page program time is 200 microseconds and the typical block erase time is 700 microseconds. With 2,048 hidden bits in 16 pages of a block, the 5,000 PE cycles will take $(0.2 * 16 + 0.7) * 5,000/1,000 = 19.5$ seconds. The throughput will be about 105 bits/second. This is the ideal case which

71

does not include program data transfers and microcontroller overhead. The hiding throughput will also be higher if we use a smaller number of PE cycles for stressing, or if we use smaller groups.

In order to read the hidden information, one needs to obtain per-bit program times using partial programming. The characterization speed depends on the number of partial programs, M, used in the decoding algorithm. For reading hidden bits (decoding), we only need to perform partial programs until more than half of the bits flip. In our experiment, M for decoding is around 30, and it takes around 3.63 seconds to characterize 16 pages, which contain 2,048 hidden bits. Therefore, the read throughput is about 564 bits/second. The read throughput will be higher if the hiding scheme uses a smaller number of Flash bits to encode each hidden bit.

For a detailed analysis to detect hidden bits (see 5.3.4), one needs to obtain a complete program time distribution with a large M. In our testbed, it takes 612.6 seconds to characterize a block using $M = 1,200$ even if we ignore data transfer from the microcontroller to the host computer and processing time on the host. A 4Gbit Flash memory chip has 4,096 blocks, so obtaining the complete program time distribution of the whole chip will take around 29 days. Higher capacity chips will take even more time to characterize for detection and decoding. For comparison, simply reading the digital content from the 4Gbit Flash chip will take approximately 4 minutes. Therefore, fully characterizing the entire Flash chip without knowing where hidden information is located is quite time consuming.

5.3.4 Detectability

The previous subsection shows that the per-bit program time in Flash memory can be controlled sufficiently to reliably store hidden information. Here, we discuss whether an attacker with access to a Flash chip can detect the existence of hidden information. In essence, the question is whether variations in Flash memory characteristics due to information hiding can be distinguished from variations due to normal use.

The proposed information hiding scheme uses per-bit program time, which is not visible from the digital content in a Flash memory device. Also, the hiding operation does not change normal Flash functions; users can still read, erase, and write Flash memory in the expected manner. Therefore, the hidden information cannot be detected from the inspection of digital content. Instead, an attacker needs to rely on checking the analog properties of the Flash memory.

The following list summarizes the steps that an attacker needs to take in order to analyze the analog properties, and in particular, the timing properties, of Flash memory.

1. Check for anomalies in timing of normal Flash operations.

2. Pick pages/blocks for more detailed analysis.

3. Collect per-bit program time for a selected page.

4. Analyze the per-bit program time distribution of a page.

5. Repeat Steps 2 to 4.

In order to determine whether a Flash chip contains hidden information or not, an attacker can start by checking the timing of normal Flash operations

73

such as per-page program time and per-block erase time, which can easily be obtained from normal operation. If these operations do not show any anomaly – their timing is within the range of timing characteristics for normal use – then the attacker needs to obtain and analyze per-bit program time by picking a page for detailed analysis, collecting per-bit program times through partial programs, and then running an analysis. If there is no way to identify suspicious pages and blocks from normal operations, in the worst case, the attacker will need to perform the detailed analysis for every single page in Flash memory, which will take a long time.

In the rest of the subsection, we will discuss each step that the attacker needs to take and whether the information that is hidden can be detected in each step.

Anomalies in Normal Flash Operations

Stressing a Flash chip may affect the analog characteristics of normal memory operations such as page read time, page program time, and block erase time. If these characteristics change significantly due to our scheme, an attacker could use that to detect the existence of hidden information. Therefore, we first study the impact of information hiding and normal Flash use on the page read time, page program time, and the block erase time.

Using the Micron 4Gbit chips, we tested six hiding PE cycle counts (625, 1,250, 2,500, 5,000, 7,500, and 10,000) and five normal PE cycle counts (0, 32, 64, 128, 256) on 4 different chips. On each chip, we used 20 blocks, each containing 64 pages. Because we hide data once every fourth pages, only 16 pages within each block are used to hide information. A normal PE cycle is performed by

74

writing randomly generated data to every page in a block, then erasing that block, simulating wear from normal usage.

To study the impact of information hiding on the page read time, we measured the time to read pages (after performing an erase) when they were fresh as well as after 5,000 hiding PE cycles. The read times were virtually identical before and after the hiding stress, showing that the read time would not be a good indicator for the existence of hidden information.

Figure 5.13 shows the program times for individual pages in two blocks from one chip, one fresh block and the other with hidden information. As shown in the figure, even though our hiding algorithm only uses every fourth page in a block, there is no visible pattern in per-page program time. The figure also shows that the program time of a page shows distinct values. The distribution between the distinct program times may change as a page wears out with PE cycles. However, we found that the possible program time values for each chip stay the same across the range of stress levels in both normal usage and information hiding cases.

Figure 5.14 shows the program time distributions across four chips for three different stress levels: fresh, 5,000 hiding PE cycles, and 32 normal PE cycles. The figure again shows that the program time falls into a small set of distinct values even though there are more distinct values across 4 chips. More importantly, pages with and without hidden information share the same set of program time values. Also, unlike per-bit program time, the experimental results show that the page program time does not change significantly with stress, at least for the particular 4Gbit chips that we tested. This is likely due to the fact that the page program time is determined by the control circuit based on the
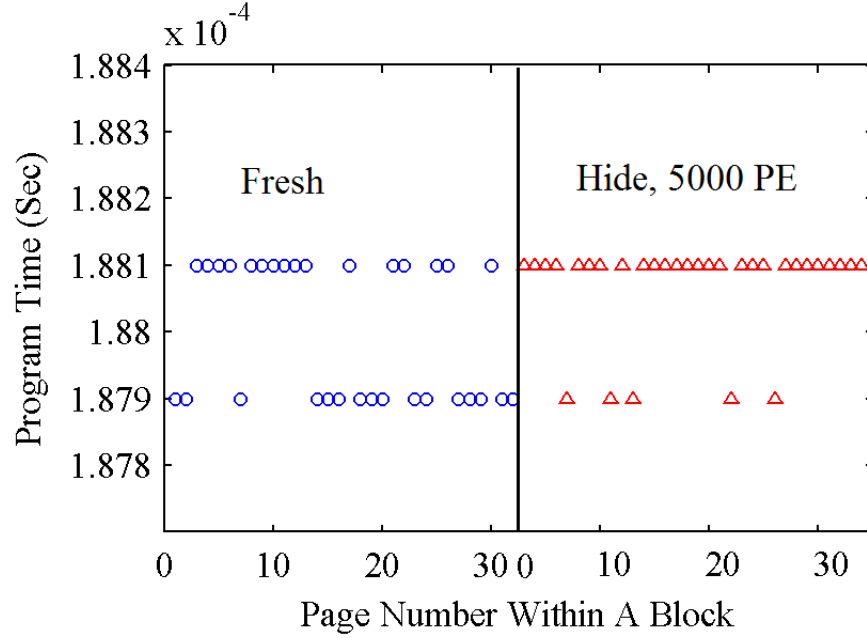
Figure 5.13: Program time for pages within a block.

slowest bit within a page. Therefore, each page's program time by itself does not show whether the page has hidden information or not.

Figure 5.15 and Figure 5.16 illustrate the block erase time distribution within a chip and across 4 chips, respectively. Similar to the program time, the erase time also falls into a few distinct levels, which are common across different stress levels. On the other hand, the figures show that the erase time tends to increase as the stress level increases. As a result, blocks with hiding stress are more likely to have a long erase time compared to fresh block without any stress. In that sense, the erase time may be used to distinguish fresh pages from blocks with hidden bits. However, because both normal PE cycles and hiding PE cycles increase the erase time, it is unclear how to distinguish blocks with hidden information from blocks with normal PE stress based on the erase time distribution (see Figure 5.16). We also found that there exist fairly large chip-
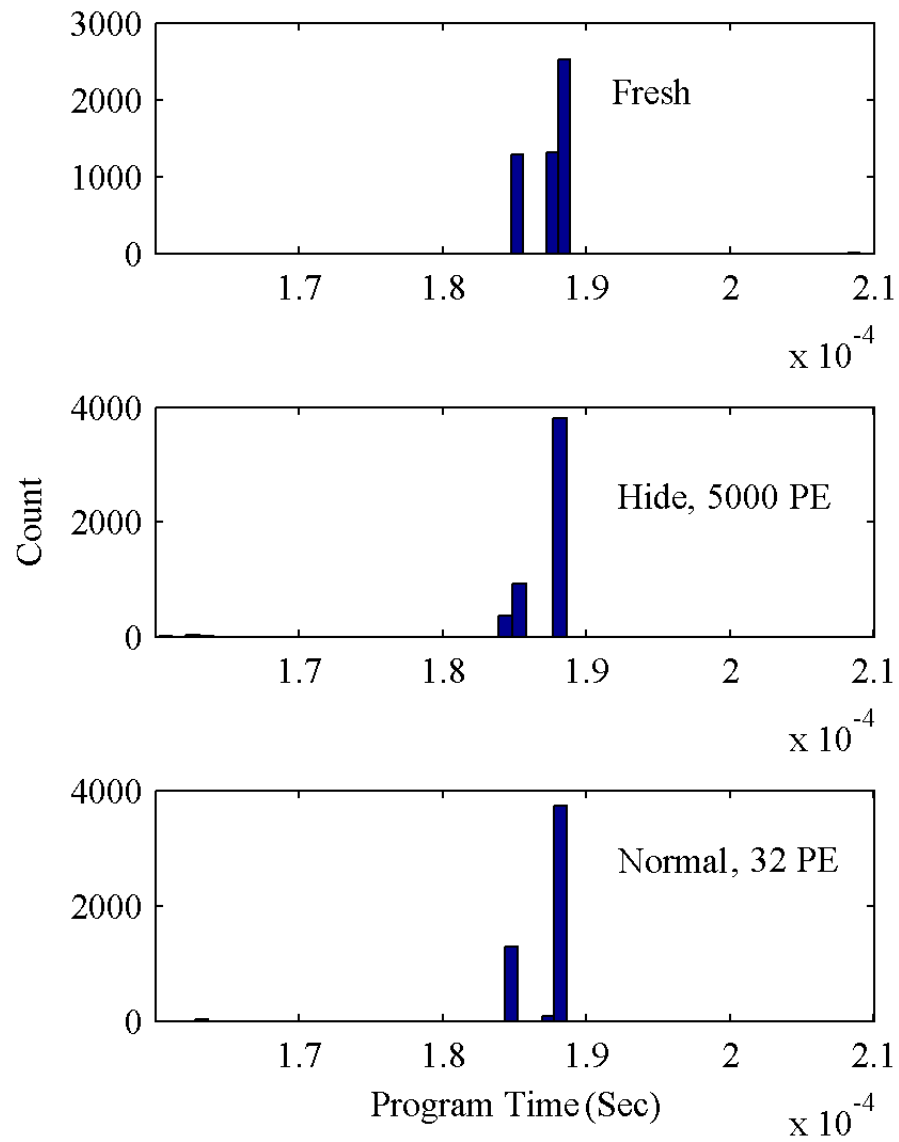
Figure 5.14: Program time histogram for three stress levels.
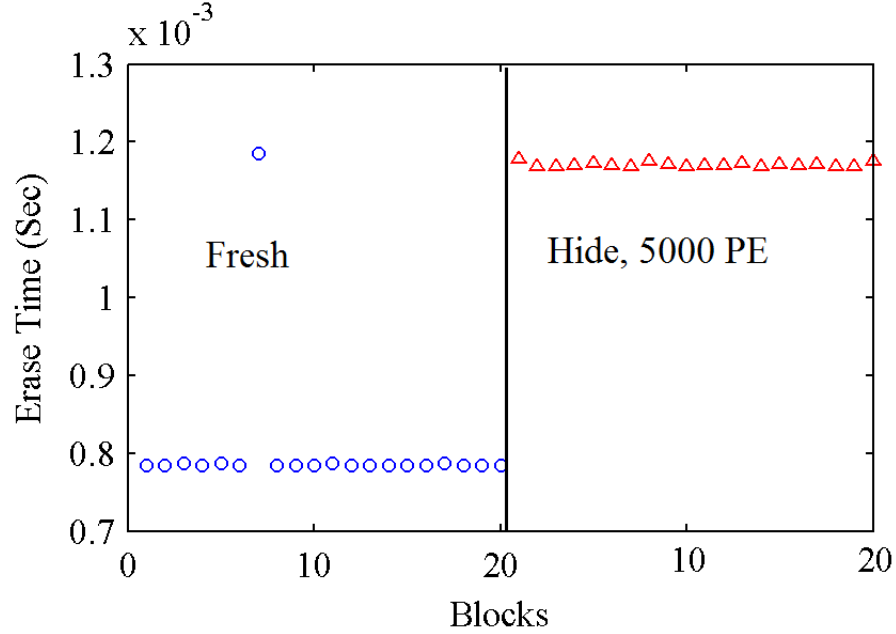
Figure 5.15: Erase time for 20 blocks within a chip.

to-chip variations. For example, some fresh chips may have over 50% of blocks that show a long erase time even without any PE stress.

The experimental results so far show that there is no obvious pattern in program time and erase time distributions to distinguish pages or blocks with hidden information from pages or block with normal PE stress. Yet, it may be possible that there exists a pattern that is difficult to detect in human eyes. To further study detectability of hidden information based on normal Flash operations timings, we tried a support vector machine (SVM) to predict whether a page or a block has hidden information. A support vector machine is a machine learning model that is widely used to recognize patterns and classify data sets. We used libsvm, a popular SVM software package [32].

For the SVM experiments, we constructed multiple data sets using pages/blocks with hidden information as well as pages/blocks with normal
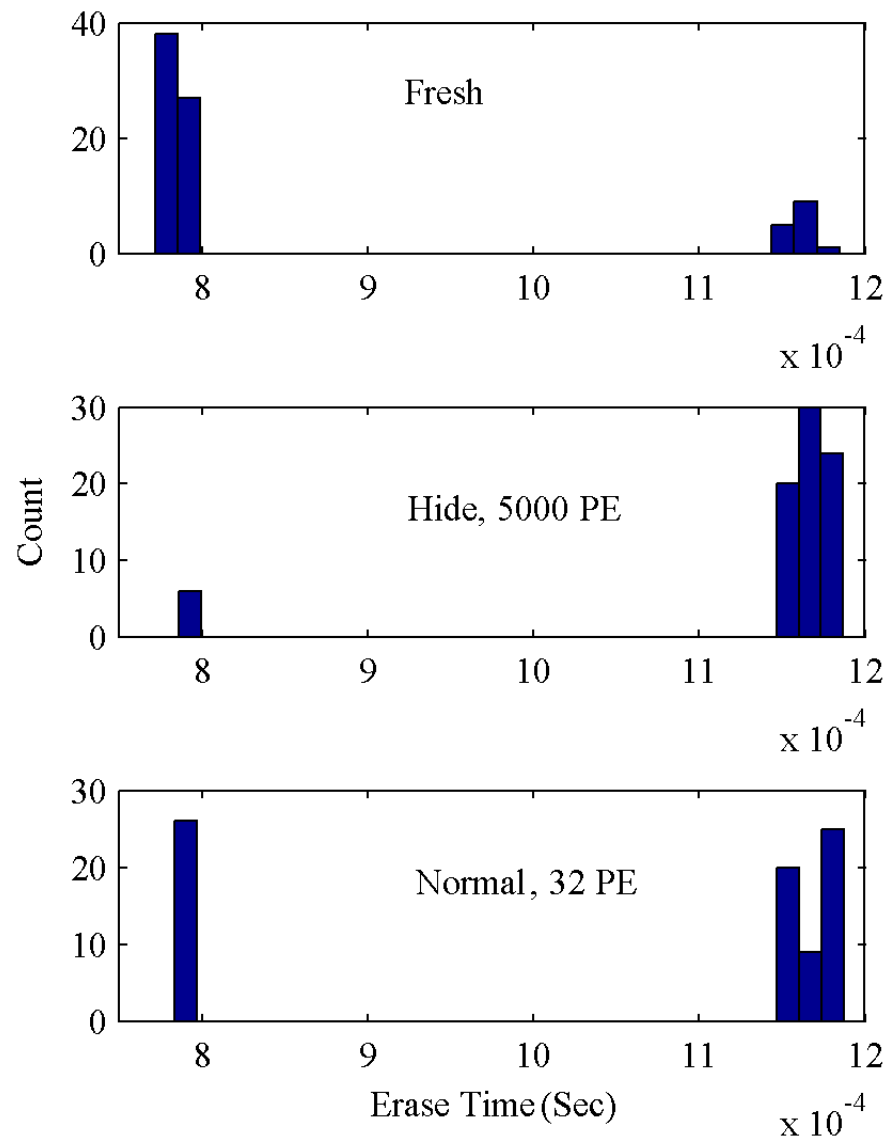
Figure 5.16: Erase time histogram for three stress levels (across 4 chips).
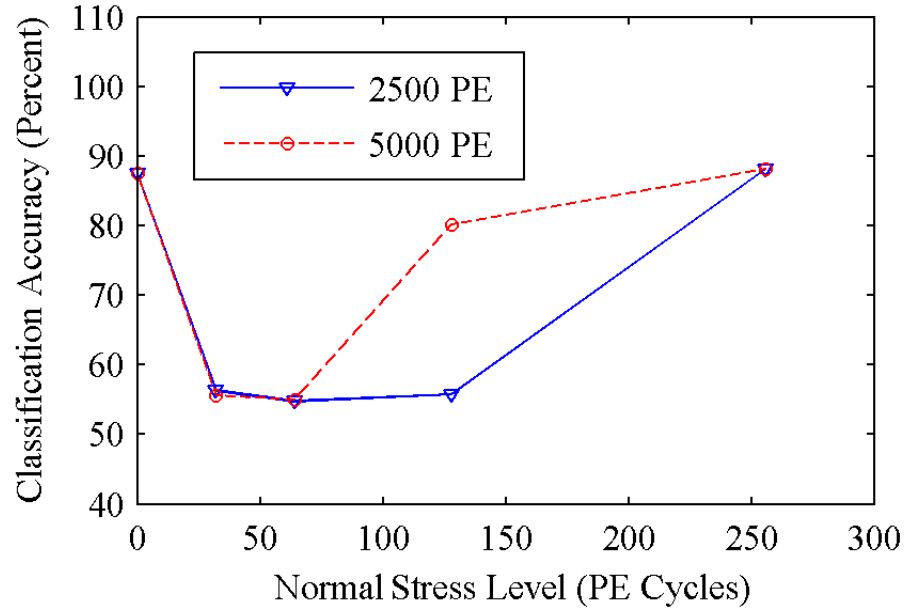
Figure 5.17: SVM accuracy for detecting hidden information (per-page analysis).

stress, combining data from one hiding stress level and one normal stress level. We used two hiding stress levels (2,500 and 5,000 PE cycles) and five normal stress levels (0, 32, 64, 128, 256 PE cycles), collected from 4 Flash chips. Then, for each data set, the SVM was trained with data from 3 chips and then tested on data from one remaining chip. This construction represents an idealistic scenario for an attacker. In practice, the attacker will need to consider all possible stress levels for both normal uses as well as hiding, which will add more variations.

Figure 5.17 shows the prediction accuracy when the SVM is given the program time and erase time for each page individually without a notion of blocks. The SVM performs relatively well when distinguishing fresh pages and pages with hiding stress. However, the accuracy drops significantly when comparing pages with hiding stress and pages with moderate levels of normal PE stress.
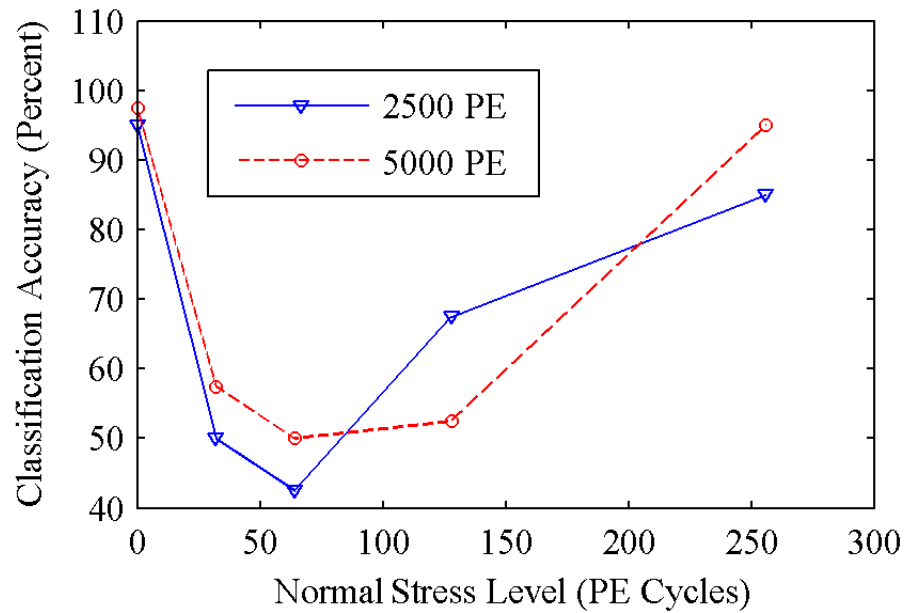
Figure 5.18: SVM accuracy for detecting hidden information (per-block analysis).

In such cases, the accuracy is not much better than random guesses (50%). The SVM performs better again for cases with high normal stress levels beyond 128 PE cycles because the normal stress exceeds the hiding stress level. In essence, the results suggest that the SVM can distinguish pages with different stress levels, but not pages with hiding stress and normal stress.

Figure 5.18 shows the accuracy of another SVM construction where the input vectors were organized by block. In this design, each input is a vector of program and erase time pairs for each page within a block, allowing the SVM to see a pattern within a block instead of handling each page separately. The goal of this SVM is to identify blocks with hidden information. The accuracy of this SVM was similar to that of the per-page SVM. The SVM could distinguish more stressed blocks from less stressed blocks, but not the hiding stress from the normal stress.

While not shown here, we also tested cases where data from all stress levels were combined together to form a large data set. We found that dealing with multiple stress levels significantly reduces SVM prediction accuracy for both the page-granularity analysis and the block-granularity analysis. The SVM predictions were no better than random guesses.

The experimental results so far show that it is difficult to distinguish pages/blocks with hiding stress from pages/blocks with normal stress even on one particular Flash model (Micron 4Gbit). In practice, an adversary will also need to deal with diversity and variations among multiple Flash manufacturers and models, which will make detecting hidden bits even more difficult.

In fact, we found that analog characteristics of Flash memory varies significantly from model to model. For example, we tested 2Gbit Flash chips from Micron, which have an identical specification with the 4Gbit chips except for the capacity. Surprisingly, the 2GBit chips, although only a generation apart from the 4Gbit chips, showed a markedly different behavior compared to the 4Gbit chips. For 2Gbit chips, the PE stress had little impact on block erase time while noticeably changing page program time. In essence, the 2Gbit chips showed the opposite type of behavior as the 4Gbit chips where the erase time shows a significant shift. In both cases, we still found that it is difficult to distinguish the impact of hiding stress from that of normal stress.

The significant variations across Flash models imply that an attacker will need to build and train an SVM model for each Flash chip model in order to use the SVM for determining the existence of hidden data on a particular chip. Obviously, this would require a significant investment on the part of the attacker. Even then, as we have shown above, there is no guarantee that an SVM model

using normal Flash operations will be able to determine the existence of hidden data with a high probability.

Page Selection and Per-Bit Program Time Collection

The study of normal Flash operations shows that an adversary cannot simply determine whether a Flash chip has hidden information or not based on measurements of normal Flash operation times. In essence, the hiding stress cannot be effectively distinguished from normal PE stress. As a result, an attacker needs to perform a more detailed analysis on per-bit program times in an attempt to determine the existence of hidden data, which we will discuss next.

To perform the detailed analysis of each page, the attacker will have to characterize each page. However, characterizing per-bit program time for every page is quite a time-consuming process. As discussed in Section 5.3.3, a 4 Gbit Flash memory chip requires around 29 days to characterize. For larger chips, which are common today, the per-bit characterization will take even longer.

To avoid expensive characterization of every page, an attacker may be able to use normal Flash operation times to select candidate pages for the detailed analysis. For example, for the 4Gbit Micron chips, an attacker may consider blocks with a higher erase time to be more likely to have hidden information. However, the study in the previous subsection suggests that pages and blocks with hiding stress can be hidden by stressing other blocks on the chip with a moderate number of normal PE cycles.
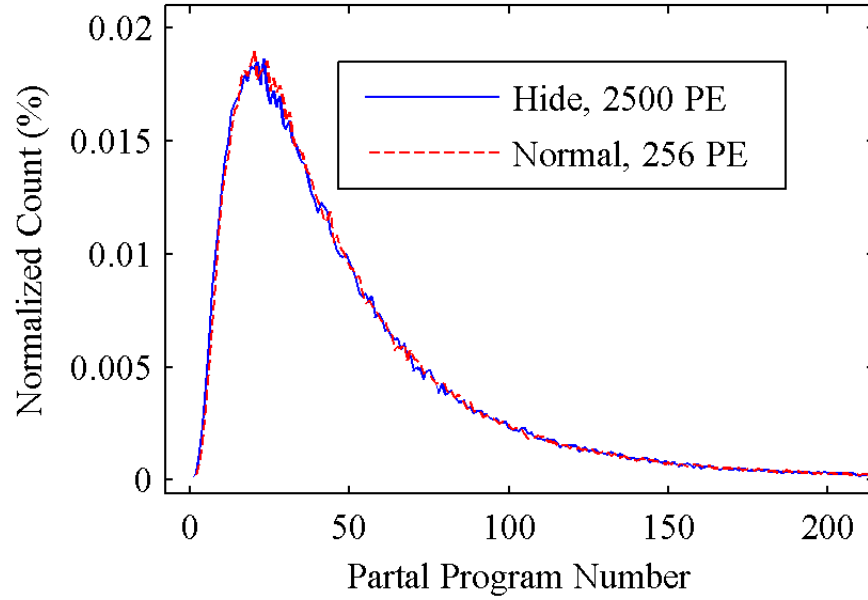
Figure 5.19: Partial program number distribution curve averaged over 5 blocks.

Per-Bit Program Time Analysis

A more detailed detectability analysis involves analyzing the partial program time distribution for bits within a page. In normal usage, the bits are programmed 0s and 1s randomly over time. In the hiding scheme, some bits are always programmed 0s and others are always programmed 1s. However, the hiding scheme does not cause an obvious bimodal distribution due to large intrinsic variations of bits in a page. Figure 5.19 shows the partial program time distribution averaged over 5 blocks. It can be seen that they are very similar to each other.

To statistically analyze the distributions, we turned to support vector machines again. To train an SVM for the per-bit analysis, we prepared pages across 2 different hiding PE stress levels (2,500 and 5,000) and 8 different normal wear

stress levels (32, 64, 128, 256, 512, 1,024, 2,048, and 4,096 PE cycles). We used 5 blocks on each chip, 16 pages per block, for a total of 80 pages per chip, at each stress level; i.e. on one chip, there are 80 pages with a hidden message stressed at 2,500 hiding PE cycles, 80 pages with a hidden message stressed at 5,000 hiding PE cycles, 80 pages without hidden data stressed 32 normal PE cycles, and so on. We characterized pages across 15 different chips. Each page represents a data point in the SVM. The SVM had access to the complete raw data for each page: the vector representing a page and an entry for each bit, with the entry's value as the partial program time.

We then grouped the data from all chips into multiple sets, combining one hiding stress level and one normal stress level. For example, one data set comprises the hidden data with 2,500 hiding PE cycles and the data with 128 normal PE cycles, another data set used 5,000 PE hidden data and 4,096 normal PE cycles, and so on, with a data set for each combination of hiding and normal PE cycles.

For each data set we labeled the hidden pages and non-hidden pages appropriately, trained the SVM with data from chips 1-10, and then used the resulting SVM to predict data from chips 11-15. Overall prediction accuracy of the SVM on test data from chips 11-15 is shown in Figure 5.20 and Figure 5.21.

Each data set is represented by a point in Figure 5.20. Normal PE stress level is shown on the X-axis. The data sets sharing 2,500 hiding PE stress are connected by a solid line; the data sets sharing 5,000 hiding PE stress are connected by a dashed line. Accuracy is shown on the Y-axis.

Overall accuracy is slightly better than random (50%) for all data sets, with
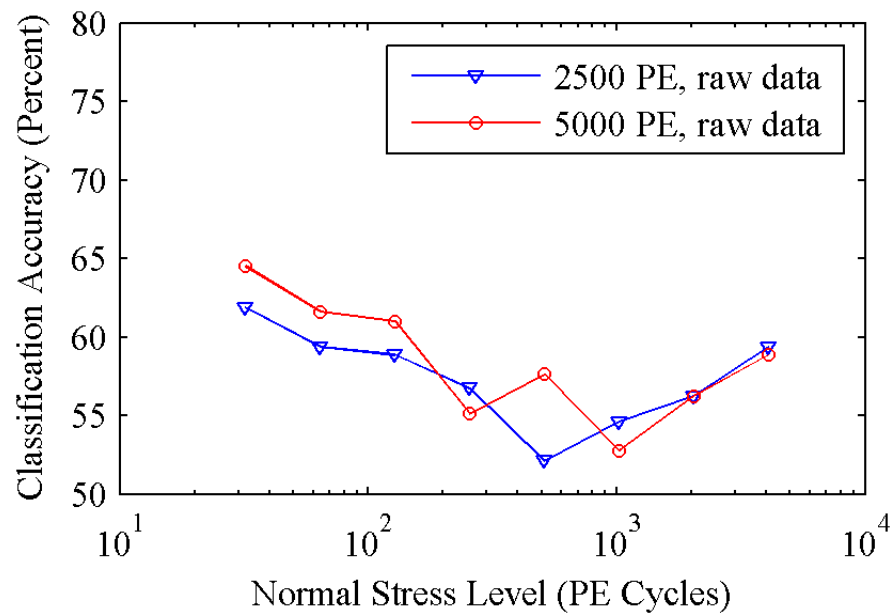
Figure 5.20: SVM accuracy for detecting pages with hidden information (using raw data).
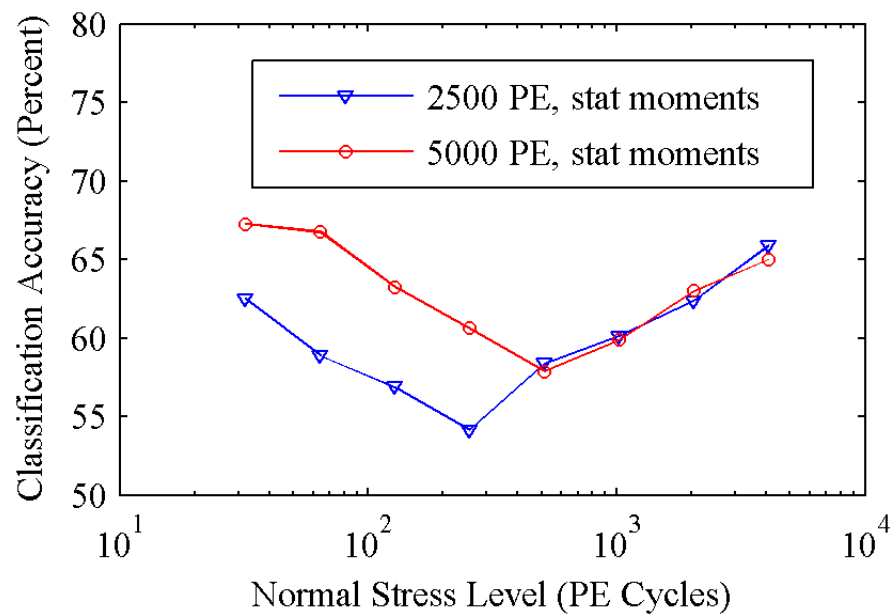
Figure 5.21: SVM accuracy for detecting pages with hidden information (using statistical moments).
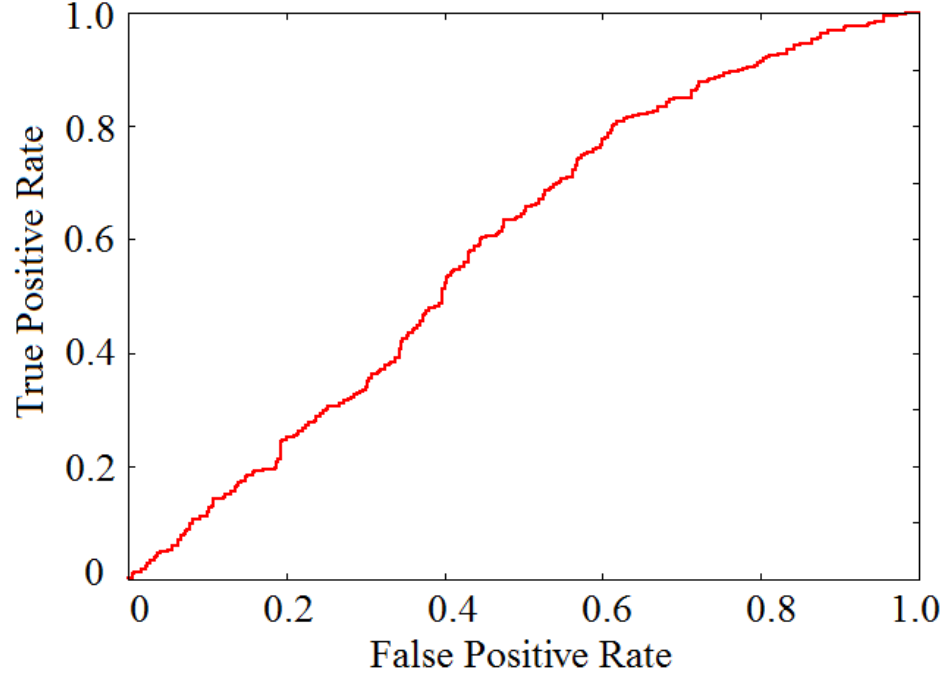
Figure 5.22: Receiver operating characteristic curve for data set including 2500 hiding PE and 128 normal PE stresses.

increased accuracy near the extremes of normal PE stress cycles. This matches the expectation that a given page with a certain hiding PE stress level looks similar to a page with a certain normal PE stress level. The further the normal PE stress level varies from the matching hidden PE stress level, accuracy should increase.

The data sets in Figure 5.21 show the SVM accuracy using a different representation for page characteristics. Instead of using the partial program count for every single bit in a page, a page was summarized by several statistical parameters: minimum, maximum, average, variance, skew, and kurtosis. We can see that prediction accuracy is similar to the SVM using the raw bit-level data.

Figure 5.22 shows a more detailed analysis of the SVM accuracy using the
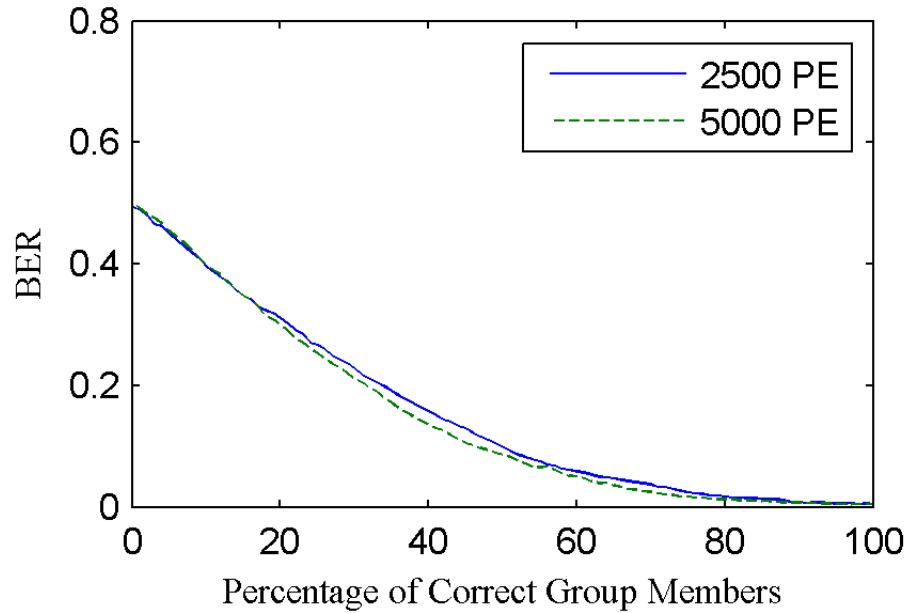
Figure 5.23: BER as a function of the percentage of correct group members.

data set for 2,500 hiding and 128 normal stresses levels. The receiver operating characteristic (ROC) curve plots the true positive rate versus the false positive rate, and gives an indication of how accurate the SVM prediction is, for a given false positive rate. The graph shows that the SVM prediction cannot achieve a high true positive rate without incurring a large percentage of false positives.

We also note that detecting hidden information is likely to be even more difficult in practice. For example, the hiding scheme may only use a subset of a page instead of every bit. Also, a classifier such as an SVM will need to deal with multiple stress levels together. We found that SVM accuracy is lower when a data set contains multiple stress levels.

5.3.5 Retrieval without the Hiding Key

Without the hiding key, one can still attempt to extract the hidden information. By estimating (through random guessing if necessary) which bits are grouped together, an attempt at extraction could reveal data if enough of the estimate is correct. Figure 5.23 shows the bit error rate versus the percentage of correctly guessed group bits.

With a large enough group and page size, it is difficult to correctly guess enough of the group members. For our group size of 128, the probability that 10% (13) of the bits in a randomly selected group of 128 bits belong to the desired group is approximately $\binom{128}{13} * (1/32)^{13}$; or 0.5%. As there are 32 groups of 128 bits in a 4,096 bit page, each bit has a 1/32 chance of being in the desired group. Even at 10%, the bit error rate is approximately 0.4. The chance of guessing 20% of the bits in a randomly selected group drops precipitously; it is 7.3e-11%. In addition, an attacker would have to try several group sizes.

Group size is a security parameter that one can adjust in order to provide greater or lesser protection against brute force group selection.

5.3.6 Erase Tolerance

To test the erase tolerance of the scheme, we deliberately stress the chip after hiding information on the chip. For this post-hiding stress, we program every bit of the page to 0, in order to put the maximum stress on the bits. The influence of post-hiding stress on the BER versus the number of PE cycles performed after hiding information is shown in Figure 5.24. From the figure, we can see that
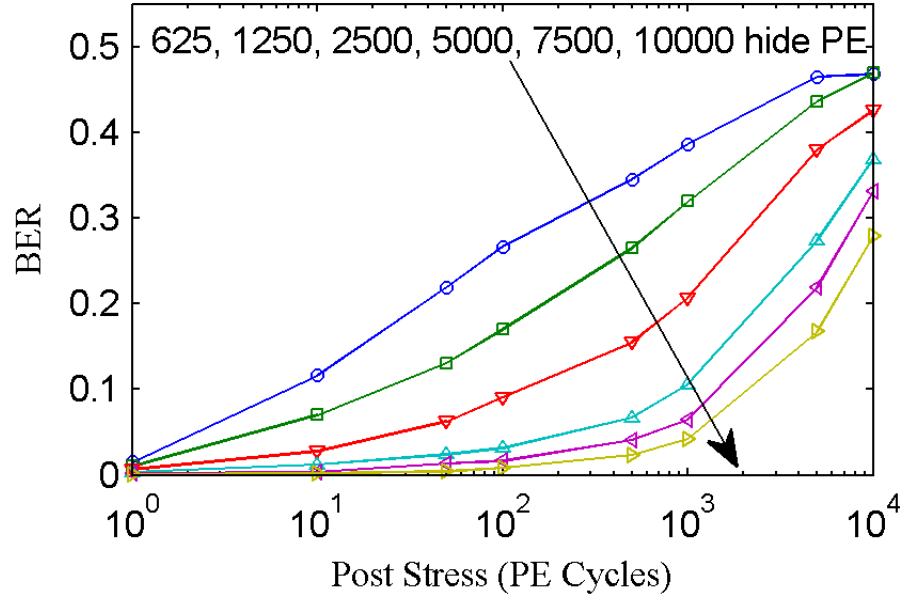
Figure 5.24: Influence of post hiding PE cycles.

the BER increases as the post PE stress level increases. However, the BER of hidden information is quite reasonable, even after hundreds of post PE cycles. For example, with 5,000 hiding PE cycles, the BER is less than 10% even after 500 post-hiding stress cycles.

5.3.7   Different Flash Models

To ensure that our scheme applies more generally, we tested several different Flash memory models (shown in Table 5.1). On all of the chips, we were able to successfully hide and recover information. We noticed that chips from the same manufacturer tend to perform similarly. For the Micron 2Gbit chips, 5 chips are tested using 10,000 hiding PE stress and 128-bit groups. The mean BER for these five chips is 0.0030. The maximum BER and minimum BER are 0.0041 and 0.0016, respectively. Chips from different manufacturers perform

differently. The tested Hynix chip has a similar BER, 0.0021, as the Micron chips in the same experiment. However, for the Hynix chips, page 0 is different from other pages in a block and, in the decoding process, a different threshold Th is needed to convert the average program time into the final binary bit for this page. The tested Numonyx chip has a very large gap for the group averages with the correct hiding key, making its BER 0 in our experiment.

We also included a multi-level cell (MLC) chip in our testing, as these chips are commonly used. MLC chips map multiple bits to each memory cell. As a result, one needs to know the mapping of bits to Flash cells to selectively stress certain cells. For the Micron MLC chip we tested, we only used the upper page in a pair of pages (as specified from the datasheet). We programmed 0 to the bits which we want to stress and 1 to the rest of the bits. Then, we programmed all of the bits to 1. Interestingly, we found that bits within a page split into a fast group and a slow group in this MLC chip, and only the faster programming bits worked for information hiding. The MLC chip required significantly fewer PE cycles to achieve the same level of BER compared to the SLC chips. For example, we used 2,000 PE cycles for our experiments and got a BER of zero – there was a large gap between the more stressed and less stressed groups.

5.4   Related Work

This section briefly summarizes prior work in steganography technologies and hardware security functions, and discusses how they are related to the information hiding technique in this paper.

5.4.1 Steganography

With the advent of information technology, digital steganography has become the subject of considerable study.

A large body of work has focused on hiding information within digital files, such as images, videos, audio files, text, and others [33, 34, 35]. These schemes usually hide data in unused meta-data fields, or by exploiting noise in the digital content itself; i.e. altering colors slightly in an image or frequency components in an audio file. In all cases the hidden data is tied to the data in the digital file. A recent proposal [36] takes a different approach: using the fragmentation pattern of digital files in a file system as a covert channel, avoiding tampering with the digital content itself. However, hidden data is still innately tied to the existence of a digital file. Also, modifying hard drive firmware has been investigated as a potential way to hide information [37]. Data is hidden in sectors marked as unusable at the firmware level (instead of the OS or filesystem level), which renders the sectors inaccessible to most software and complicates recovery, as it is difficult to tell legitimately bad sectors from ones used for hiding.

Our proposed scheme for Flash memory shares the concept of exploiting noise to hide data, in the sense that intentionally created biases are hidden in inherent variations in Flash program time. However, unlike the above methods, in which hidden information depends upon plainly visible digital files, our information hiding scheme uses analog properties of Flash. As a result, hidden information is decoupled from the digital content and instead tied to a physical object. The use of physical properties makes detecting, copying, or erasing of hidden information difficult because it requires detailed and time-consuming analog measurements.

Some steganographic techniques hide information where it is not encoded in plainly visible digital files. For example, there exist methods to hide information in the noise of wireless and optical transmissions by modifying the physical layer protocol [38, 39, 40]. Our work presents a new way to hide information in Flash memory. Unlike previous techniques, which often require special tools or modifications to existing protocols, the proposed information hiding technique can be applied to Flash memory chip through a standard interface without any hardware modification.

To make the steganographic functions available in the embedded domain, Stanescu et al. proposed to use an FPGA to efficiently process steganographic algorithms [41]. Our technique gives embedded platforms the ability to hide info within the device at a level not visible to the file system, and requires no additional hardware, as Flash memory is common on embedded platforms.

5.4.2   Flash Based Security

We hide a message in the per-bit program times of Flash memory. Given the popularity of Flash memory in computing systems, there have been studies on analog characteristics of Flash memory [31]. While we have gained insight from the previous work, it primarily focuses on using analog variations to build more efficient computing systems rather than enhancing security.

Recently, there have been proposals to use noise and variations in Flash memory for security by generating true random numbers and unique chip fingerprints [5, 7]. We use the partial programming technique that was proposed by the previous study. However, this paper proposes a completely new appli-

cation of Flash memory in the context of information hiding instead of random number generation and fingerprinting.

5.4.3   Physical Unclonable Functions

Physical Unclonable Functions (PUFs) exploit process variation to provide unique fingerprints for logic circuits [3]. Special circuits are built that vary their output depending on the process variation specific to one instance of the chip. This work is related to PUFs in the sense that we exploit physical properties and process variations for security purposes. However, unlike PUFs, our information hiding scheme uses process variations to hide information instead of generating device-specific fingerprints and keys. Also, our information hiding technique can be applied using standard Flash chips and does not require any custom circuitry.

CHAPTER 6

CONCLUSION

In this work, we show that unmodified Flash chips are capable of providing two important security functions: high-quality true random number generation and the provision of many digital fingerprints. Using thermal noise and random telegraph noise, random numbers can be generated at up to 10Kbit per second for each Flash bit and pass all NIST randomness tests. An authentication scheme with fingerprints derived from partial programming of pages on the Flash chip show high robustness and uniqueness. The authentication scheme was tested over 24 pages with 24 different instances of a Flash chip and showed clear separation. A Flash chip can provide many unique fingerprints that remain distinguishable in various temperature and aged conditions. Both random number generation and fingerprint generation require no hardware change to commercial Flash chips. Because Flash chips are ubiquitous, the proposed techniques have a potential to be widely deployed to many existing electronic device though a firmware update or software change.

We also demonstrate a technique to hide information using the program time of individual bits in Flash memory. Program time is an analog characteristic of Flash and is not visible from digital content, does not affect normal memory operation, and survives Flash data erasure. Measuring program time can be done over the standard Flash interface (with no hardware modification) via partial programming. Using groups of bits to store one bit of payload allows the technique to effectively hide information robustly with low bit error rates, and makes detection difficult to prove unless one knows the hiding key. Without the key, measuring analog characteristics of the Flash chip reveals nothing

that cannot be explained by normal wear or manufacturing variation. We note that retaining a copy of the entire analog characteristics of the Flash memory requires a large amount of time.

BIBLIOGRAPHY

[1] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[2] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, DTIC Document, 2001.

[3] G.E. Suh and S. Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *Proceedings of the 44th Design Automation Conference*, 2007.

[4] Patrick Koeberl, Jiangtao Li, Anand Rajan, Claire Vishik, and Wei Wu. A practical device authentication scheme using sram pufs. In *Trust and Trustworthy Computing*, pages 63–77. Springer, 2011.

[5] Pravin Prabhu, Ameen Akel, Laura Grupp, Wingkei Yu, G.E. Suh, Edwin Kan, and Steven Swanson. Extracting Device Fingerprints from Flash Memory by Exploiting Physical Variations. In *Trust and Trustworthy Computing*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011.

[6] N.F. Johnson and S. Jajodia. Exploring Steganography: Seeing the Unseen. *Computer*, 1998.

[7] Yinglei Wang, Wingkei Yu, G. Edward Suh, and Edwin Kan. Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[8] MJ Kirton and MJ Uren. Noise in solid-state microstructures: A new perspective on individual defects, interface states and low-frequency (1/) noise. *Advances in Physics*, 38(4):367–468, 1989.

[9] Hideaki Kurata, Kazuo Otsuga, Akira Kotabe, Shinya Kajiyama, Taro Osabe, Yoshitaka Sasago, Shunichi Narumi, Kenji Tokami, Shiro Kamohara, and Osamu Tsuchiya. Random telegraph signal in flash memory: Its impact on scaling of multilevel flash memory beyond the 90-nm node. *Solid-State Circuits, IEEE Journal of*, 42(6):1362–1369, 2007.

[10] C Monzio Compagnoni, Michele Ghidotti, Andrea L Lacaita, Alessandro S Spinelli, and Angelo Visconti. Random telegraph noise effect on the programmed threshold-voltage distribution of flash memories. *Electron Device Letters, IEEE*, 30(9):984–986, 2009.

[11] Sung-Min Joe, Jeong-Hyong Yi, Sung-Kye Park, Hyungcheol Shin, Byung-Gook Park, Young June Park, and Jong-Ho Lee. Threshold voltage fluctuation by random telegraph noise in floating gate nand flash memory string. *Electron Devices, IEEE Transactions on*, 58(1):67–73, 2011.

[12] Donald E Knuth. The art of computer programming, volume 1: fundamental algorithms. *Addison-Wesley, Reading, Mass*, 1973.

[13] Kenichi Abe, Akinobu Teramoto, Shigetoshi Sugawa, and Tadahiro Ohmi. Understanding of traps causing random telegraph noise based on experimentally extracted time constants and amplitude. In *Reliability Physics Symposium (IRPS), 2011 IEEE International*, pages 4A–4. IEEE, 2011.

[14] John H Scofield, Nick Borland, and DM Fleetwood. Temperature-independent switching rates for a random telegraph signal in a silicon metal–oxide–semiconductor field-effect transistor at low temperatures. *Applied Physics Letters*, 76(22):3248–3250, 2000.

[15] Open NAND Flash Interface. http://onfi.org.

[16] Texas Instruments. Omap mobile processors, 2013.

[17] Berk Sunar, William J Martin, and Douglas R Stinson. A provably secure true random number generator with built-in tolerance to active attacks. *Computers, IEEE Transactions on*, 56(1):109–119, 2007.

[18] Charles W Odonnell, G Edward Suh, and Srinivas Devadas. Puf-based random number generation. *In MIT CSAIL CSG Technical Memo*, 481, 2004.

[19] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. Fpga-based true random number generation using circuit metastability with adaptive feedback control. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 17–32. Springer, 2011.

[20] C. Dike G. Cox and D. J. Johnston. Intels digital random number generator, 2011.

[21] EE Times.com. U.s.: Fake parts threaten electronic market, 2010.

[22] Sosfakeflash.

[23] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *Advances in cryptology-Eurocrypt 2004*, pages 523–540. Springer, 2004.

[24] D Boning and J Chung. Statistical metrology: Understanding spatial variation in semiconductor manufacturing. In *Microelectronic Manufacturing Yield, Reliability, and Failure Analysis II: SPIE 1996 Symp. On Microelectronic Manufacturing*, 1996.

[25] Keith A Bowman, Steven G Duvall, and James D Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Solid-State Circuits, IEEE Journal of*, 37(2):183–190, 2002.

[26] Sani R Nassif. Modeling and forecasting of manufacturing variations. In *Statistical Metrology, 2000 5th International Workshop on*, pages 2–10. IEEE, 2000.

[27] Pappu Srinivasa Ravikanth. *Physical one-way functions*. PhD thesis, Massachusetts Institute of Technology, 2001.

[28] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[29] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *VLSI Circuits, 2004. Digest of Technical Papers. 2004 Symposium on*, pages 176–179. IEEE, 2004.

[30] Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. Initial sram state as a fingerprint and source of true random numbers for rfid tags. In *Proceedings of the Conference on RFID Security*, volume 7, 2007.

[31] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. Characterizing Flash Memory: Anomalies, Observations,

and Applications. In *Proceedings of the 42nd International Symposium on Microarchitecture*, 2009.

[32] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[33] Abbas Cheddad, Joan Condell, Kevin Curran, and Paul Mc Kevitt. Digital Image Steganography: Survey and Analysis of Current Methods. *Signal Processing*, 2010.

[34] N. Provos and P. Honeyman. Hide and Seek: An Introduction to Steganography. *IEEE Security & Privacy*, 2003.

[35] F.A.P. Petitcolas, R.J. Anderson, and M.G. Kuhn. Information Hiding - A Survey. *Proceedings of the IEEE*, 1999.

[36] Hassan Khan, Mobin Javed, Syed Ali Khayam, and Fauzan Mirza. Designing a cluster-based covert channel to evade disk investigation and forensics. *Computers & Security*, 30(1):35 – 49, 2011.

[37] Iain Sutherland, Gareth Davies, and Andrew Blyth. Malware and Steganography in Hard Disk Firmware. *Journal in Computer Virology*, 2011.

[38] A.M. Mehta, S. Lanzisera, and K. Pister. Steganography in 802.15.4 Wireless Communication. In *Proceedings of the 2nd International Symposium on Advanced Networks and Telecommunication Systems*, 2008.

[39] Krzysztof Szczypiorski and Wojciech Mazurczyk. Steganography in IEEE 802.11 OFDM Symbols. *Security and Communication Networks*, 2011.

[40] P.R. Prucnal, M.P. Fok, K. Kravtsov, and Zhenxing Wang. Optical Steganography for Data Hiding in Optical Networks. In *Proceedings of the 16th International Conference on Digital Signal Processing*, 2009.

[41] D. Stanescu, V. Stangaciu, I. Ghergulescu, and M. Stratulat. Steganography on Embedded Devices. In *Proceedings of the 5th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2009.