Microcontroller TRNGs Using Perturbed States of NOR Flash Memory Cells

Prawar Poudel, Biswajit Ray, *Member, IEEE*, and Aleksandar Milenkovic, *Senior Member, IEEE*

Abstract—This paper introduces a new technique that perturbs split-gate NOR Flash memory cells and extracts randomness of read noise to generate true random numbers. Flash memory cells exhibit threshold voltage fluctuations during read operations caused by thermal noise and random telegraph noise effects. Recent proposals demonstrate how these inherent properties of Flash memory cells can be used to create true random numbers in modern NAND Flash memories. However, they cannot be directly applied to NOR Flash memories in microcontrollers that have different architecture, improved data retention, high endurance, and are not as susceptible to noise as high-density NAND Flash memories. The proposed technique is experimentally demonstrated and evaluated using a family of commercial microcontrollers. The evaluation shows that it enables extraction of high-throughput random sequences that pass the NIST statistical tests. Advantages of the proposed technique are as follows: (a) it does not require any special hardware and/or interface modifications, (b) it is robust, cost-effective, and high-throughput, (c) it is entirely implemented in software, and (d) it is flexible and can be tailored to work in low-end microcontrollers that are often resource- or cost-constrained.

Index Terms—Real-time and embedded systems, measurement techniques

1 Introduction

Random number generators (RNGs) are used in computer systems to generate random cryptographic keys, in devices for electronic gambling, and in many other applications where randomization and fairness are required, e.g., in statistical sampling, computer simulation, and video games. Two main approaches to generate random numbers are Pseudo-Random Number Generators (PRNG) and True Random Number Generators (TRNGs). PRNGs rely on deterministic algorithms that produce numbers that are in practice indistinguishable from truly random numbers, provided an attacker cannot guess their starting state or seed. They are very efficient, easy to implement in software, and have high throughput. However, PRNGs rely on a truly random seed and may be thus vulnerable to cryptanalytic attacks [1]. For example, a highly sophisticated attack on slot machines in casinos was recently uncovered and reported in news [2]. This attack that resulted in significant losses in gambling industry exploits deterministic nature of PRNGs and an attacker's ability to guess the seed. TRNGs are typically hardware modules that generate truly random numbers from stochastic physical processes, such as, radioactive decay, photoelectric effect, single photon optical processes, Brownian motion, clock jitters, or electronic noise. Unlike PRNGs that use deterministic algorithms, TRNGs produce random numbers that cannot be predicted, but typically suffer from lower throughput and require special hardware resources to tap into random physical processes. The importance of TRNGs has been recognized by the industry, and many recent high-end processors include dedicated hardware resources for generating true random numbers [3].

There have been a number of academic proposals for TRNGs and many of them are amiable for implementation in electronic devices. One group of proposed TRNGs targeting ASICs and FPGAs exploits clock jitter. For example, Maiti et al. [4] proposed using jitter from multiple ring oscillators to generate random numbers in ASICs. Random frequency jitter is also used in an ASIC TRNG proposed by Tang et al. [5]. This approach was extended by Johnson et al. [6] in their tunable TRNG that extracts jitter from beat frequency detectors in Xilinx FPGAs. Another group of proposed TRNGs exploits metastability of bi-stable circuit elements, such as flip-flops. For example Majzoobi et al. [7] introduced an FPGA-based TRNG that exploits metastability of logic states with adaptive feedback control. Wieczorek et al. [8] introduced dual metastability TRNG that harnesses the time taken by a bi-stable circuit to resolve its state as a source of randomness. Another group of proposals focuses on extracting randomness from memory structures. For example, Holcomb et al. [9] used the state of SRAM cells on power-up to produce device fingerprints; these fingerprints contain sufficient entropy to generate 128-bit true random numbers as well. Tehranipoor et al. [10] proposed a TRNG that utilizes DRAM remanence effect. Other proposals exploit noise in Flash memory chips [11], [12]. Several other recent TRNG proposals exploit switching instability in emerging memory technologies such as resistive random access (RRAM) [13] and spin transfer-torque magnetic memory (MRAM) [14].

All these proposals represent a great advancement in the field of TRNGs and meet the required application specific needs. However, several important obstacles to most of these proposals remain. First, some of them are applicable only to newly designed ASICs and do not address the need for TRNGs in many existing chips. Next, many of these proposals rely on dedicated on-chip circuitry to extract randomness, which increases on-chip area, and thus chip cost. Some of the proposals have a relatively low throughput or may require cost-prohibitive or inconvenient operations, such as power-up or reset cycles. Emerging Internet-of-Things applications often rely on low-end resource- and cost-constrained microcontrollers that do not include built-in support for true random number generation. Yet, these devices may produce "high value" information that need to be communicated securely. Hence, designing TRNGs that can work in low-end devices is very important.

This paper introduces a new technique for generating true random numbers by exploiting read noise of perturbed NOR Flash memory cells. NOR Flash memories are random access non-volatile memories used in modern microcontrollers to store programs and data. Common read operations from Flash memory involve sensing of Flash cells threshold voltage, which is in either erased state (logic '1') or programmed state (logic '0'). The actual threshold voltage fluctuates due to random telegraph noise and thermal noise effects. A typical Flash memory design ensures sufficient noise margins to guarantee correct read operations. In the proposed method, we use partial programming of Flash memory words to induce perturbed states of Flash memory cells. In this state, the threshold voltage is very close to the reference read voltage. This way, the state of the Flash cell is uncertain during read operations—it can be either logic '1' or logic '0', depending on read noise. We characterize split-gate NOR flash memory found in a family of commercial microcontrollers and describe algorithms for inducing perturbed states and identifying strongly perturbed Flash memory cells (Section 3). These cells are then used in the proposed algorithm for generating random numbers (Section 4). The proposed algorithm is demonstrated and evaluated on a family of commercial microcontrollers from Texas Instruments, MSP430F5438 (Section 5).

• *The authors are with the Electrical and Computer Engineering Department, University of Alabama in Huntsville, Huntsville, AL 35899.*
E-mail: {pp0030, biswajit.ray, milenka}@uah.edu.

Manuscript received 20 Mar. 2018; revised 10 Aug. 2018; accepted 17 Aug. 2018. Date of publication 20 Aug. 2018; date of current version 22 Jan. 2019.
(Corresponding author: Aleksandar Milenković.)
Recommended for acceptance by P. Faraboschi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2018.2866459

0018-9340 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

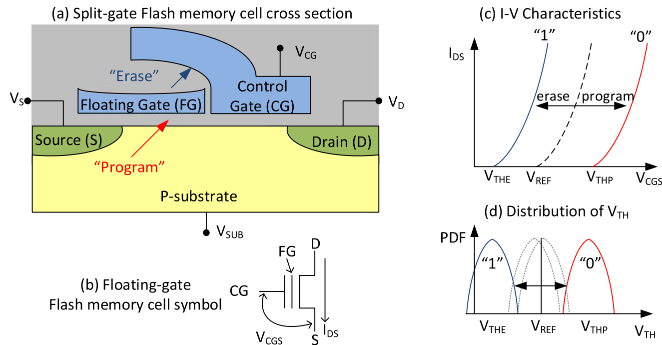


Fig. 1. (a) Cross-section of split-gate Flash memory cell. (b) Floating-gate transistor symbol. (c) Current-voltage characteristics of a split-gate Flash memory cell (V_{THE}—threshold voltage for the erased state, V_{THP}—threshold voltage for the programmed state, V_{REF}—reference threshold voltage); (d) Probability density function of the threshold voltage for erased, programmed, and perturbed cells.

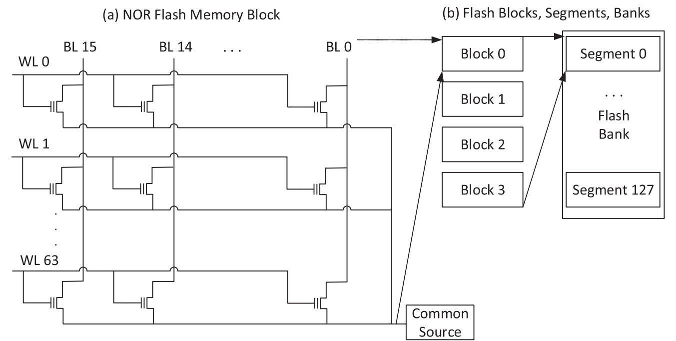


Fig. 2. (a) NOR Flash memory block architecture. (b) Flash memory organization: blocks, segments, and banks.

The proposed technique offers several advantages compared to the existing TRNGs. First, utilizing Flash memory in microcontrollers makes this proposal widely applicable and affordable. Second, the proposed random number extraction technique does not require any hardware change, specific circuits, or system-prohibited (or privileged) operations, such as hardware reset or power on cycle. Hence, it can be implemented in software as a library function. Third, the proposed algorithm produces high-quality random bits that pass the NIST statistical tests and achieves a very good throughput. Fourth, the proposed algorithm is parameterized and can be tuned to work in resource-constrained microcontrollers with minimal memory or processing power.

The Flash memory based TRNG introduced in this paper is not the first one of its kind. Wang et al. [11] proposed a TRNG that exploits random telegraph noise in disturbed NAND Flash memory cells. Their algorithm relies on sophisticated pre-conditioning steps that include repeated partial programming and periodical refresh operation of Flash memory pages. Ray and Milenkovic [12] used repeated program operation on the same page to bring NAND Flash memory cells into disturbed state and their algorithm exploits read noise. However, both of these proposals focus on high-density NAND Flash memories that are designed to achieve high capacity. These proposals cannot be directly applied to NOR Flash memories that are routinely used in modern microcontrollers. NOR Flash memories in microcontrollers typically have improved data retention, high endurance, and are not as susceptible to noise as high-density NAND Flash memories. To the best of our knowledge, this is the first work to demonstrate practical TRNGs that can be easily implemented in software in low-end microcontrollers.

The key contributions of this paper are as follows: (a) it introduces a mechanism for perturbing NOR Flash memory cells and explores design trade-offs; (b) it introduces an algorithm for extracting random bits from read noise in perturbed NOR Flash cells; (c) it implements the proposed algorithm on a family of microcontrollers and explores its sensitivity to algorithm parameters; and (d) it evaluates the proposed TRNG for quality of random bits and effectiveness of implementation.

2 Background

2.1 Flash Memory Cell Structure

Fig. 1a shows a cross-section of a split-gate Flash memory cell that keeps one bit of information. Fig. 1b shows a transistor symbol for a floating-gate Flash memory cell. Each cell contains two gates, the floating gate (FG) and the control gate (CG), arranged to lie above the transistor channel. This arrangement differs from a traditional stacked Flash memory cell, where the control gate is placed directly above the floating gate that in turn sits right above the transistor channel. In a split-gate Flash memory cell, the floating gate occupies a portion of the area between the drain and source and is completely electrically isolated. The control gate has a unique shape that covers a portion of the area between the drain and source as well as a portion of space above the floating gate [15], [16].

A Flash memory cell can be in one of two states, erased which is equivalent to logic '1' and programmed which is equivalent to logic '0'. Two major operations are performed to change the state of Flash memory cells: *program* that charges the floating gate with electrons and *erase* that discharges the floating gate. These operations require high voltages and are carried out through the oxide as shown in Fig. 1a. To program a split-gate Flash memory cell, a large voltage is applied to the source terminal $(V_S \sim 10V, V_{CG} \sim 2V,\ V_D \sim 0.5V)$ inducing source-side hot carrier injection (SSI) that results in electron injection on the floating gate ("Program" arrow). The electrons on the floating gate reduce the voltage between the control gate and source, thus increasing the threshold voltage $(V_{TH} = V_{THP})$ as shown in Fig. 1c. To erase Flash memory cells, a large positive voltage is applied on the control gate $(V_{CG} \sim 12V,\ V_S = V_D = 0V)$ to remove electrons from the floating gate via Fowler-Nordheim tunneling ("Erase" arrow). The removal of electrons decreases the threshold voltage $(V_{TH} = V_{THE})$ as shown in Fig. 1c.

A read operation from Flash memory involves applying a read voltage on the control gate, $V_{CG} = V_{READ} \sim 3V$, and a sense voltage on the drain, $V_D = V_{SENSE} \sim 2V$, and sensing the threshold voltage. An erased cell will conduct the current and that is sensed as logic '1' and a programmed cell will not conduct the current and that is sensed as logic '0'.

2.2 NOR Flash Memory Organization

NOR Flash memories are designed to allow random access through full address and data buses, fast reads, and low standby power. However, they have lower storage capacity, lower density, and longer erase and program times than NAND Flash memories. NAND Flash memories are designed for high-capacity and low-cost storage solutions, but they do not provide a random access address bus.

Flash memory cells are organized in multiple two-dimensional arrays known as Flash memory blocks. Fig. 2a shows a typical NOR Flash memory block [17]. The control gates of cells in each row are electrically connected through a line called Word Line (WL). The drains of all cells in a single column are electrically connected through a single line called Bit Line (BL). The source terminals of all cells are also electrically connected to a common source terminal. The number of cells in each row defines a word size and in our case we assume 16-bit words. Thus, a Flash memory block illustrated in Fig. 2a has capacity of 64 16-bit words or 128 bytes. Blocks are further organized into segments. In our example shown in Fig. 2b, a segment includes 4 blocks for a total of 256 words.

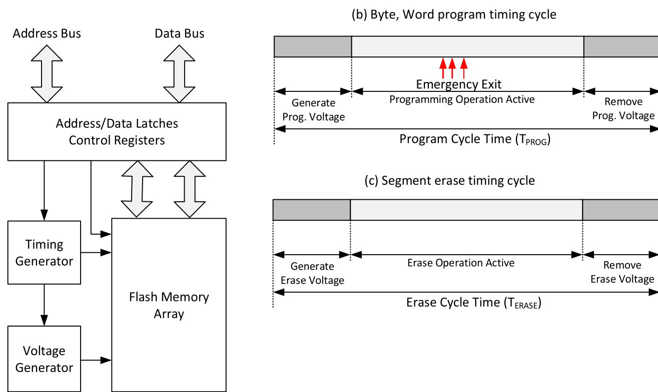


Fig. 3. (a) Block diagram of a Flash memory module (b) Program cycle time; (c) Erase cycle time.

Segments are further organized into Flash memory banks, e.g., a 64 KB Flash memory bank includes 128 segments.

2.3 Programmers View of Flash Memory

Modern microcontrollers rely on the NOR Flash memories as one of the most popular nonvolatile memories to store code and constant data values. The default mode of operation is read, where Flash memory behaves as a ROM. However, Flash memories are often in-system programmable through a Flash memory controller. The Flash memory can be programmed at a byte or a word level through the controller, whereas the smallest unit that can be erased is an entire segment. Some Flash memory controllers support mass erase of an entire Flash memory bank. Each Flash bit can be programmed from logic '1' to logic '0' individually, but to reprogram a bit from logic '0' to logic '1' requires an erase cycle.

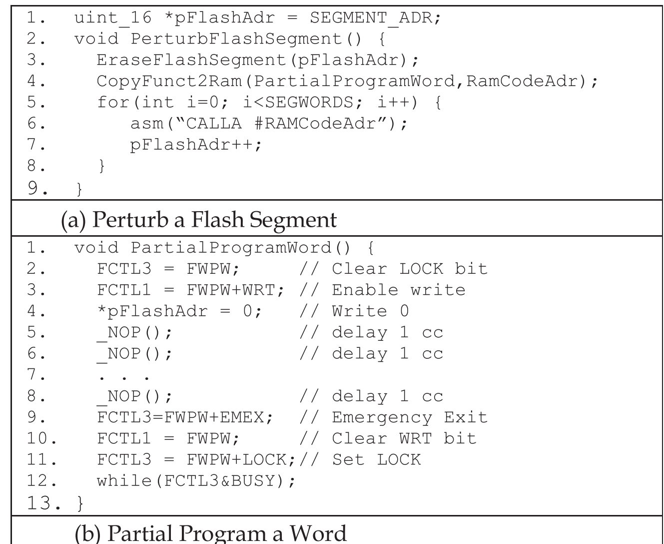
Fig. 3a shows a block diagram of a Flash memory module in a microcontroller that includes a Flash memory array with one or more banks and a Flash controller. The controller includes voltage generators that supply voltage levels needed for program and erase operations as well as timing generators that control duration of operations that span multiple clock cycles. A Flash memory program (write) operation can be initiated by a microcontroller code that is running from within the Flash memory itself (albeit from a different region) or from RAM. When initiating a write operation from within the Flash memory, the processor is typically halted until the program operation completes. When initiating a program operation from RAM, the processor can continue executing the code from RAM, provided it does not interact with the Flash memory bank that contains a location currently being programmed. Fig. 3b shows a typical program cycle that encompasses times to bring up voltage generators, perform the program operation, and remove programming voltages to allow Flash memory to resume operation in its default mode. The total programming time in a microcontroller used in our study is $T_{PROG} \approx$ 64-85 μs. An erase cycle is initiated by a dummy write to an address belonging to a segment to be erased. Similar to program operations, erase operations can be initiated from within the Flash memory halting the processor, or from within the RAM when the processor can continue execution. Fig. 3c shows a typical erase cycle that encompasses times to bring up voltage generators, perform segment erase operation, and remove programming voltages to allow Flash memory to resume operations in its default mode. The total segment erase time in a microcontroller used in this study is $T_{ERASE} \approx$ 23-35 ms.

2.4 Perturbed Flash Memory Bits

In normal operation, a Flash memory cell is stable in either programmed or erased state. A read operation determines the state of a memory cell. Read is performed at a byte or a word granularity as all the bits in a word share the same WL. The sensing process involves application of a sense voltage $V_{SENSE} \sim 2V$ on the bit lines and a read voltage $V_{CG} \sim 3V$ on the selected word line (the common source is grounded, $V_S = 0V$). All other unselected WLs are biased at low voltage so that all the unselected memory cells are turned off. With this biasing arrangement, current will flow from the bit line to the ground through the bits whose threshold voltage is less than the read voltage. The read reference voltage (V_{REF}) is set in between the erased state and programmed state distributions, so that there is enough of noise margin to correctly identify the bit states as shown in Fig. 1d.

During sensing operation a cell current may fluctuate due to thermal noise or random telegraph noise (RTN), which in extreme cases may cause false identification of the erased cell as the programmed cell or vice versa. In this paper, we refer to a combination of thermal noise and RTN as "read noise" since both of them coexist in Flash memories. Thermal noise is white noise generated by the thermal agitation of the charge carriers and it exists in all electronic devices. RTN is usually caused by the random trapping and de-trapping of charge carriers at semiconductor-oxide interfaces. The amplitude of noise fluctuation in current/voltage due to RTN is inversely proportional to the gate area.

To bring a Flash memory cell into a perturbed state (neither programmed nor erased), we need to ensure that its threshold voltage is close to the reference voltage (Fig. 1d). In this state, the read noise will be a determining factor for reading logic '0' or logic '1'. In this paper, we use partial programming to intentionally bring a cell into a perturbed state. A method proposed by Ray and Milenkovic [12] to use the program disturb phenomena—repeated programing of the checkerboard pattern into the Flash memory pages—works well in high-density NAND flash memories, but does not appear to work on NOR Flash memories we have analyzed. In addition, exploiting the program disturb phenomena requires repeated program cycles which takes time and wears-down selected region of the Flash memory.



```
1.  uint_16 *pFlashAdr = SEGMENT_ADR;
2.  void PerturbFlashSegment() {
3.    EraseFlashSegment(pFlashAdr);
4.    CopyFunct2Ram(PartialProgramWord,RamCodeAdr);
5.    for(int i=0; i<SEGWORDS; i++) {
6.       asm("CALLA #RAMCodeAdr");
7.       pFlashAdr++;
8.    }
9.  }
```

(a) Perturb a Flash Segment

```
1.  void PartialProgramWord() {
2.    FCTL3 = FWPW;      // Clear LOCK bit
3.    FCTL1 = FWPW+WRT;  // Enable write
4.    *pFlashAdr = 0;    // Write 0
5.    _NOP();            // delay 1 cc
6.    _NOP();            // delay 1 cc
7.    . . .
8.    _NOP();            // delay 1 cc
9.    FCTL3=FWPW+EMEX;   // Emergency Exit
10.   FCTL1 = FWPW;      // Clear WRT bit
11.   FCTL3 = FWPW+LOCK;// Set LOCK
12.   while(FCTL3&BUSY);
13. }
```

(b) Partial Program a Word

Fig. 4. Perturbing Flash memory bits: (a) C subroutine for perturbing a Flash segment. (b) C subroutine for partial programming a Flash word.

3 Perturbing Flash Memory Bits

To bring a Flash memory segment into a perturbed state we use a method outlined in Fig. 4a. First, a selected Flash segment is erased—all bits are set to logic '1' (line 3). To bring each word up into a perturbed state, it is partially programmed. However, partial Flash programming is not possible when the code is running from within the Flash memory because the processor is stalled during

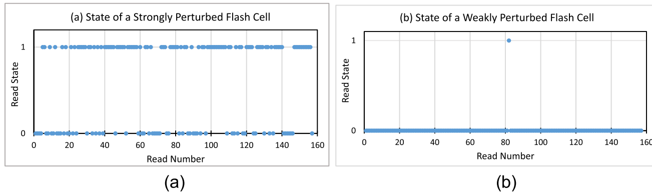


Fig. 5. State of perturbed Flash memory bits during consecutive reads; (a) strongly perturbed bit; (b) weakly perturbed bit.

the entire programming cycle. Consequently, we first copy a function that performs partial programming into RAM (line 4). To partially program a Flash segment, we invoke the *PartialProgramWord ()* function from RAM for each word in a segment (line 6).

The partial programming function shown in Fig. 4b configures Flash controller registers for a program operation (lines 2-3) and then writes 0x0000 to the selected word (line 4). A certain number of NOP instructions (lines 5-8) follows this instruction before an emergency exit command is issued to the Flash controller (line 9). Note: to reduce a memory footprint NOP instruction can be replaced by a loop with controllable software delay. In the microcontroller used in our study any program or erase operation can be stopped before its normal completion by setting the emergency exit bit EMEX in a Flash controller register. Setting this bit stops the active operation and resets the Flash memory controller. All operations cease and the Flash memory returns to read mode. The busy bit is used to determine the end of the emergency exit cycle (line 12), when it is safe to return to the caller running from within the Flash memory. The state of the partially programmed word is unpredictable. Note: the exact mechanism to partially program a Flash segment may differ among different microcontrollers. Often, duration of program and erase operations can be controlled by a hardware timer that can then trigger premature end of these operations.

Fig. 5 shows states of two representative perturbed bits as a function of consecutive reads. A word in Flash memory is partially programmed and then it is read a number of times. Fig. 5a shows a state of a strongly perturbed Flash cell with a threshold voltage very close to the reference read voltage. The state of the cell appears to be fluctuating randomly over time. Fig. 5b shows a state of a weakly perturbed Flash cell. This cell's threshold voltage is shifted toward the programmed state side of distribution, but once in the observed period it is read as erased. Our goal is thus to bring Flash cells into a strongly perturbed state, identify such Flash cells, and use them for generating random numbers.

Ideally, we would like to have as many Flash memory cells in a perturbed state as possible, with their threshold voltage as close to the read reference voltage as possible. This way, algorithms for extracting true random numbers will achieve higher throughput and require fewer memory locations. To achieve this goal we need to determine appropriate moment to issue an emergency exit operation. The programming cycle starts when the write operation is performed (Fig. 4b, line 4). Each NOP instruction requires exactly one clock cycle allowing us to fine tune software delay in search for an optimal moment to execute emergency exit. The software delay expressed in processor clock cycles is a function of the number of NOP instructions and the time to execute emergency exit. Another tunable parameter is the processor clock frequency. The microcontroller used in our study enables full software control over processor clock frequency. When the processor runs at a higher clock frequency, we need to spend more clock cycles waiting before aborting the Flash programing cycle, but we can do it with a finer time resolution. Note: an exact implementation of the Flash memory controller and its clock-generating portion is not disclosed, so we cannot know the exact relationship between the processor clock frequency and the logic inside the Flash controller

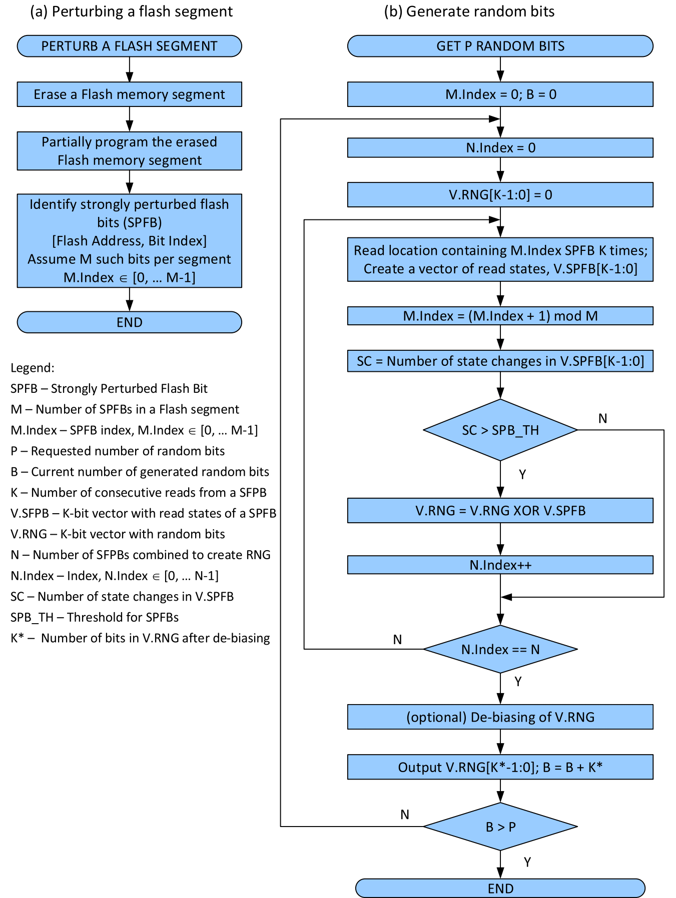


Fig. 6. Algorithms for (a) perturbing a Flash memory segment and (b) generating random bits from a perturbed Flash segment.

responsible to abort the ongoing programming operation when the EMEX command is issued. The results of this analysis will be presented later.

4 Algorithm for TRNG

In order to extract read noise characteristics through a digital interface, our first step is to bring a Flash memory segment into a perturbed state. Fig. 6a shows a sequence of preconditioning steps to bring a segment into such a state. The selected Flash memory segment is first erased and then partially programmed word-by-word as described in Section 3. The partial programming is carried out in such a way to maximize the number and quality of perturbed bits. To speed up random number generation, an initial profiling is performed during preconditioning to identify strongly perturbed flash bits. We have observed that strongly perturbed Flash bits (SPFBs) do not lose their properties over a very long period and are a good source of randomness. The profiling is done by repeated reading of Flash memory words within a perturbed segment and recording the number of state changes for each perturbed bit. If the number of state changes of a perturbed bit exceeds a certain threshold, the bit is marked as a strongly perturbed Flash bit. Here we will assume that we have M such bits within a segment, each with its own index, $M.Index \in [0 \ldots M-1]$.

Fig. 6b shows the proposed algorithm for extracting random numbers from a perturbed Flash segment. The algorithm traverses the segment words containing strongly perturbed Flash bits. Each such a word is read repeatedly K times, and a K-bit vector, V.SPFB, containing read values of a particular SPFB is created. Though these bits are initially marked as the SPFB, we again count the number of state changes and ensure that they maintain their

TABLE 1
TI MSP430F5438 Flash Characterization

Clock rate [Hz]	Partial program time		Average number of perturbed bits ± stdev	
	#CPU Cycles	[μs]	Strong	Total
1,048,576	27	25.75	0 ± 0.0	0 ± 0.0
	28	26.70	10 ± 5.3	24 ± 7.5
	29	27.66	0 ± 0.0	0 ± 0.0
4,194,304	95	22.65	0 ± 0.0	0 ± 0.0
	96	22.89	2 ± 1.3	4 ± 1.6
	97	23.13	135 ± 19.1	301 ± 32.2
	98	23.37	5 ± 3.0	10 ± 5.6
	99	23.60	0 ± 0.4	0 ± 0.5
	100	23.84	0 ± 0.0	0 ± 0.0
8,388,608	195	23.25	0 ± 0.4	1 ± 0.8
	196	23.37	5 ± 2.0	11 ± 1.7
	197	23.48	38 ± 9.1	88 ± 18.9
	198	23.60	142 ± 15.4	324 ± 39.2
	199	23.72	63 ± 21.6	139 ± 44.7
	200	23.84	14 ± 3.1	33 ± 9.9
	201	23.96	2 ± 1.9	4 ± 4.1

properties before they are used for random number generation. The number of state changes in V.SPFB should exceed a certain threshold, SPB_TH, usually expressed as a fraction of the number of consecutive reads, *K*. The current vector V.SFPB is then XOR-ed with the resulting *K*-bit vector V.RNG. This operation is repeated *N* times, using first *N* distinct SPFB bits. The resulting random vector, V.RNG, is output.

Optionally, the output vector can be de-biased using the Von Neumann algorithm that takes two bits of the random sequence at a time, discards both bits if they are identical, and takes the first bit if they are different. If more random bits are required, the process continues traversing the remaining SPFB bits in the segment. Once all SPFB bits in the segment are exhausted, we reset *M.Index* to point to the first perturbed bit in the segment and repeat the process as shown in Fig. 6b. If more bits are required $(P > K)$, the process is repeated until a sufficient number of random bits is generated.

The proposed algorithm can be fine-tuned by adjusting parameters, such as *K* and *N*. These parameters impact the quality of random bits, the throughput of the proposed algorithm defined as the number of random bits generated in a time unit, as well as its complexity measured in the number of processor clock cycles spent for random number generation and the size of RAM memory needed for storing temporary variables. By adjusting these parameters, the algorithm can be tailored to run on microcontrollers with a limited size of RAM memory. In our studies we used the following parameters: $K = 1024, N = 10$.

5 Results

5.1 Experimental Setup

Our experimental setup is based on MSP430, a mixed-signal microcontroller family from Texas Instruments. The MSP430 family is built around a 16-bit processor. It integrates on a single chip the processor, Flash memory, SRAM memory, clock oscillators, and a wide range of 8-bit and 16-bit input/output peripherals, including parallel ports, timers, comparators, analog-to-digital and digital-to-analog converters, serial communication interfaces, LCD controllers, and DMAs. The MSP430 family chips that belong to different generations and models—generally higher numbered models are larger and include more features—differ in processor speed, size of memories, and the number and type of peripherals. The clock subsystem is controllable from software and supports several clock signals that can be changed or selectively turned on and off to allow for a low power operation. The Flash memory is in-system programmable and its architecture corresponds to the one described in Section 2.

In this study we used MSP430F5438 that includes 256 KB of Flash memory (4 banks each or 64 KB), 16 KB of SRAM memory, timers, serial communication interfaces, parallel ports, and an ADC controller. The experiments are conducted using the Experimenter Development Board, EXP430F5438. It features a 100-pin drop-in socket for microcontrollers allowing for quick changes of microcontroller chips.

5.2 Partial Programming Characterization

We conduct a series of experiments to explore the relationship between the number of perturbed bits in a segment, the processor clock frequency, and the duration of partial programming. To characterize the MSP430F5438 Flash memory, we consider multiple Flash segments that are pre-conditioned as described above. Each word is sequentially read *K* times, and each bit is characterized based on its state—perturbed or stable. For perturbed Flash memory bits, we record the number of state changes during *K* consecutive reads. If a perturbed bit changes its state more than $K/8$ times, we consider such a bit to be strongly perturbed; otherwise it is weakly perturbed.

Table 1 shows the results of experimental evaluation of partial programming of the MSP430F5438 Flash memory performed at room temperature of $\sim 23\,^{\circ}\mathrm{C}$. The algorithm for perturbing Flash segments is run on eight 256-word Flash memory segments, while varying the processor clock frequency and the duration of partial programming. In the first series of experiments, the processor clock frequency is set to the default $\mathrm{F_{CPU}} = 1,046,576\,\mathrm{Hz}$, and the duration of partial programming is varied in the range between 20 and 30 processor clock cycles. The reference manual indicates that programming of a Flash memory word takes between 64 and 85 μs. At this processor clock frequency, we observe a very narrow time window for perturbing Flash cells. If the duration of partial programming is above 29 processor clock cycles (which translates into 27.66 μs), all Flash memory cells in the segment are in the programmed state. Similarly, if the duration of partial programming is below 28 cycles (26.70 μs), all cells are in the erased state, leaving the delay of 28 processor clock cycles as the only delay that results in perturbed bits. The total average number of perturbed bits per segment for a tested chip is 24 (out of 4,096 bits in the segment) and 10 of these are strongly perturbed bits. The total number of perturbed bits per segment varies among different segments, ranging from 16 to 40, whereas the number of SPFBs ranges from 4 to 20.

The second series of experiments is carried out when the processor clock frequency is set to $\mathrm{F_{CPU}} = 4,194,304\,\mathrm{Hz}$. This allows a finer time resolution when controlling the duration of partial programming. The best results are achieved when the emergency exit is activated with the delay of 97 processor clock cycles, which translates to 23.137 μs. The average total number of perturbed bits per segment is 301 and the average number of strongly perturbed bits is over 135. Similar results are observed in the third series of experiments when the processor clock is further increased to $\mathrm{F_{CPU}} = 8,388,608\,\mathrm{Hz}$. Higher processor clock frequency enables more precise control of timing that results in an increased number of perturbed Flash cells. Thus, the best-case average total number of perturbed bits is 324 and the average number of SPFBs is 142 (Table 1). At this clock frequency, multiple delay periods result in perturbed Flash bits.

One may notice a discrepancy in the "optimal" duration of partial programming—it changes with changes in the processor clock frequency. It should be noted that the exact relationship between the Flash controller operation and the processor clock is not fully disclosed in chip documentation for MSP430F5438. In addition, it appears that the optimal software delay for perturbing flash segment slightly varies among different Erase-Program cycles for the single chip as well as among different chips.

TABLE 2
NIST Statistical Tests

NIST Test	Original Sequence, N = 10						Debiased Sequence, N = 5					
	SC1		SC2		SC3		SC1		SC2		SC3	
	P-Value	Propo rtion	P-Value	Propo rtion	P-Value	Propo rtion	P-Value	Propo rtion	P-Value	Propo rtion	P-Value	Propo rtion
Frequency	0.740	10/10	0.534	10/10	0.534	9/10	0.122	10/10	0.534	9/10	0.122	10/10
Block Frequency	0.740	10/10	0.740	10/10	0.350	10/10	0.534	10/10	0.069	10/10	0.035	9/10
Cumulative Sums	0.740	10/10	0.122	10/10	0.534	9/10	0.911	10/10	0.740	9/10	0.351	10/10
Runs	0.213	10/10	0.911	9/10	0.740	9/10	0.350	10/10	0.534	10/10	0.911	10/10
Longest Run	0.213	10/10	0.911	10/10	0.534	10/10	0.067	10/10	0.534	9/10	0.213	10/10
Rank	0.740	10/10	0.350	10/10	0.122	10/10	0.534	10/10	0.122	10/10	0.350	10/10
FFT	0.740	10/10	0.350	10/10	0.534	10/10	0.351	10/10	0.018	10/10	0.213	10/10
NonOverlapping T.	0.911	10/10	0.911	10/10	0.534	10/10	0.911	10/10	0.740	10/10	0.740	10/10
Overlapping Template	0.534	10/10	0.740	10/10	0.534	10/10	0.035	10/10	0.911	10/10	0.740	10/10
Universal	0.740	10/10	0.350	10/10	0.213	10/10	0.351	10/10	0.350	9/10	0.740	10/10
Approximate Entropy	0.534	10/10	0.740	10/10	0.534	10/10	0.740	10/10	0.911	10/10	0.911	10/10
RandomExcursions	–	8/8	–	5/5	–	3/3	–	5/5	–	6/6	–	7/7
Random Excursions V.	–	8/8	–	5/5	–	3/3	–	5/5	–	6/6	–	7/7
Serial	0.534	10/10	0.122	10/10	0.534	9/10	0.740	10/10	0.70	10/10	0.740	10/10
Linear Complexity	0.534	9/10	0.35	10/10	0.911	10/10	0.911	10/10	0.350	10/10	0.911	10/10

We can draw several conclusions from these experiments. First, regardless of external conditions we are able to bring each Flash memory segment into a perturbed state with a sufficient number of SPFBs. Next, by increasing the processor clock frequency we can significantly increase the number of Flash memory cells in the perturbed state; it reaches over 300 bits per segment (out of 4,096 bits), with over 100 being strongly perturbed. This by far exceeds the minimum number of strongly perturbed bits needed for our algorithm to produce quality random bits. The optimal duration of partial programming does not vary between different Flash segments in a single chip or words within a segment, but it is a function of the processor clock frequency and general state of the Flash memory. It should be noted that Table 1 shows findings for one particular sample chip. We find that other chips of the same type may differ in the optimal duration of partial programming that produces the largest number of perturbed bits. Thus, the program for perturbing a Flash segment should allow for fine-tuning of the partial programming time. On the other hand, a Flash memory segment once brought into a perturbed state remains in such a state for years – the preconditioning steps are done only once and do not depend on powering cycle. In our experiments, we have not observed any loss of characteristics over a period of 6 months. Software development tools often support an option that just a portion of Flash memory is updated when downloading a new firmware through JTAG. This means that once a Flash segment is perturbed, it can remain in use throughout the lifetime of the product, regardless of possible firmware upgrades.

To explore sensitivity to temperature variations, the setup is placed in a heated oven and a freezer and the number of SPFBs per segment is observed. The number of SPFBs varies with temperature—e.g., a Flash segment with 116 SPFBs at 23 °C has 95 SPFBs at 0 °C and 140 at 80 °C. The proposed algorithm ensures that only "good" SPFBs are used to generate random sequences and is thus robust in presence of temperature variations. SPFBs that lose their property are not used for generating random sequences. Having a pool of SPFBs $(\mathrm{M} > \mathrm{N})$ ensures that we can always find a sufficient number of "good" SPFBs.

5.3 Analysis of Randomness

To evaluate the randomness of the bits produced by the proposed Flash memory TRNG, we utilize the NIST Test Suite, a statistical package consisting of 15 tests developed to test randomness of arbitrarily long binary sequences produced by either hardware or software [18]. Each statistical test calculates a P-value that shows the randomness of the given sequences based on that test. If P-value ≥ 0.0001, then the sequence can be considered to be uniformly distributed. Note that some of the tests such as Non Overlapping Template, Random Excursions, and Random Excursions Variant consists of several individual tests. We report the least P-value out of several internal tests for them.

Table 2 shows the results for three different MSP430F5438 sample chips (SC1, SC2, SC3). For each chip, ten original random sequences (without de-biasing) and ten de-biased sequences of 1,000,000 bits are considered. The results show the P-values as well as the proportion—the number of sequences that pass the test requirements. Each test passes if at least 8 out of 10 sequences pass, excluding Random Excursion tests. The parameters of the proposed algorithm used for the original sequences in are as follows: $K = 1,024; N = 10; P = 10,000,000$. For de-biased sequences, we use $\mathrm{K} = 1,024, \mathrm{N} = 5, \mathrm{P} = 10,000,000$. We can see that all tests pass on all sample chips, which is an excellent result. One interesting question is what is the minimum N for which we can pass the NIST tests for original and de-biased sequences? As shown above when $\mathrm{N} \geq 10$, all original sequences pass the NIST tests. By lowering N, the original RNG sequences start failing individual tests. However, with the Von-Neumann de-biasing, we find that the proposed algorithm passes all the tests on all tested chips with N as low as $N = 5$.

5.4 Performance and Complexity

The throughput, defined as the number of good random bits generated in a unit of time, is a function of the algorithm parameters (N and K), the number of perturbed bits, and the processor clock frequency. The algorithm includes repeated reads from Flash memory locations that are marked to contain perturbed bits, extraction of read states from the perturbed bits, and calculation of the resulting vector as shown in Fig. 6. For $N = 10$ and $K = 1,024$ we find that our optimized algorithm implementation on MS430F5438 requires ~123 processor clock cycles per one good random bit generated. Depending on the processor clock frequency, this computational complexity translates into 8,525 random bits per second for $\mathrm{F_{CPU}} = 1,048,576\,\mathrm{Hz}\,(\sim 1\,\mathrm{MHz})$ or 68,200 bits per second for $\mathrm{F_{CPU}} = 8,388,608\,\mathrm{Hz}$. This throughput by far exceeds typical requirements for random numbers in low-end embedded systems that may require 256-bit random keys per one communication session established periodically, e.g., once every minute or hour. Whereas hardware TRNGs in microcontrollers are often able to achieve higher throughputs—e.g., produce one good random bit per clock cycle—several implementations require 100 processor clock cycles per 1 good random bit [19] which is comparable to our algorithm.

In addition to good throughput, the proposed algorithm does not require significant RAM memory resources (less than 512 Bytes

to generate numbers) and they can be even further reduced by tuning algorithm parameters (e.g., by lowering parameter *K*). In our experiments we allocated one Flash memory segment of 512 bytes that serves as a source of entropy. This segment is reserved for our algorihtm and cannot be used for other purposes. However, depending on the number of SPFBs, a portion of one segment would often be sufficient, whereas the rest can be used for code or data.

6 Conclusion

In this paper, we introduce a new technique for true random number generation that exploits read noise of partially programmed NOR Flash memory cells in microcontrollers. The proposed technique relies on a pre-conditioning algorithm to bring Flash memory bits into a perturbed state and an algorithm to efficiently generate true random numbers by reading states of strongly perturbed Flash memory cells. We describe various design trade-offs and test the randomness of generated sequences using the NIST statistical test suite. The proposed technique offers a number of advantages over the existing approaches: (a) it requires no additional hardware support, (b) it is solely implemented in software and can be easily implemented in modern microcontrollers, (c) it produces high-quality random sequences, (d) it achieves a good throughput, and (e) it can be easily tuned to work on low-end resource-constrained microcontrollers.

References

[1] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, “Cryptanalytic attacks on pseudorandom number generators,” in *Fast Software Encryption*, vol. 1372, S. Vaudenay, Ed. Berlin, Germany: Springer, 1998, pp. 168–188.

[2] B. Koerner, “Russians engineer a brilliant slot machine cheat—and casinos have no fix,” *Wired*, 06-Feb-2017. [Online]. Available: https://tinyurl.com/jc6u8rh, Accessed Mar. 2018.

[3] Intel Corporation, “Intel digital random number generator (DRNG),” 15-May-2014. [Online]. Available: https://tinyurl.com/z5cn3dy, Accessed Mar. 2018.

[4] A. Maiti, R. Nagesh, A. Reddy, and P. Schaumont, “Physical unclonable function and true random number generator: A compact and scalable implementation,” in *Proc. 19th ACM Great Lakes Symp. VLSI*, 2009, pp. 425–428.

[5] Z. Tang, X. Zhang, Y. Zhang, and L. Qi, “Portable true random number generator for personal encryption application based on smartphone camera,” *Electron. Lett.*, vol. 50, no. 24, pp. 1841–1843, Nov. 2014.

[6] A. P. Johnson, R. S. Chakraborty, and D. Mukhopadyay, “An improved DCM-based tunable true random number generator for xilinx FPGA,” *IEEE Trans. Circuits Syst. II Express Briefs*, vol. 64, no. 4, pp. 452–456, Apr. 2017.

[7] M. Majzoobi, F. Koushanfar, and S. Devadas, “FPGA-based true random number generation using circuit metastability with adaptive feedback control,” in *Cryptographic Hardware Embedded Syst.*, vol. 6917, B. Preneel and T. Takagi, Eds. Berlin, Germany: Springer, 2011, pp. 17–32.

[8] P. Z. Wieczorek and K. Golofit, “Dual-metastability time-competitive true random number generator,” *IEEE Trans. Circuits Syst. Reg. Paper*, vol. 61, no. 1, pp. 134–145, Jan. 2014.

[9] D. E. Holcomb, W. P. Burleson, and K. Fu, “Power-Up SRAM state as an identifying fingerprint and source of true random numbers,” *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1198–1210, Sep. 2009.

[10] F. Tehranipoor, W. Yan, and J. A. Chandy, “Robust hardware true random number generators using DRAM remanence effects,” in *Proc. IEEE Int. Symp. Hardware Oriented Security Trust*, 2016, pp. 79–84.

[11] Y. Wang, W. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan, “Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints,” in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 33–47.

[12] B. Ray and A. Milenkovic, “True random number generation using read noise of flash memory cells,” *IEEE Trans. Electron Devices*, vol. 65, no. 3, pp. 963–969, Mar. 2018.

[13] S. Balatti, S. Ambrogio, Z. Wang, and D. Ielmini, “True random number generation by variability of resistive switching in oxide-based devices,” *IEEE J. Emerg. Sel. Top. Circuits Syst.*, vol. 5, no. 2, pp. 214–221, Jun. 2015.

[14] Won Ho Choi, et al., “A magnetic tunnel junction based true random number generator with conditional perturb and real-time output probability tracking,” in *Proc. IEEE Int. Electron Devices Meeting*, 2014, pp. 12.5.1–12.5.4.

[15] P. Forstner, “MSP430 flash memory characteristics,” Apr. 2008. [Online]. Available: https://tinyurl.com/y8zfmp4f, Accessed Mar. 2018.

[16] A. R. Duncan, M. J. Gadlage, A. H. Roach, and M. J. Kay, “Characterizing radiation and stress-induced degradation in an embedded split-gate NOR flash memory,” *IEEE Trans. Nucl. Sci.*, vol. 63, no. 2, pp. 1276–1283, Apr. 2016.

[17] L. Crippa, R. Micheloni, I. Motta, and M. Sangalli, “Nonvolatile memories: NOR versus NAND architectures,” in *Memories in Wireless Systems*, R. Micheloni, G. Campardo, and P. Olivo, Eds. Berlin, Germany: Springer, 2008, pp. 29–53.

[18] L. E. Bassham, et al., “A statistical test suite for random and pseudorandom number generators for cryptographic applications,” National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-22r1a, 2010.

[19] Freescale, “KL82 sub-family reference manual,” Jan. 2016. [Online]. Available: https://tinyurl.com/y73b78lz, Accessed Mar. 2018.