Defense Against Flash Write Denial of Service Attacks in IoT Devices

Harmon Herring[1], Connor Jackson[1], Spencer Roth[1], Chris Sequeira[1], Ivan De Olivera Nunes[1], and Adam Caulfield[1]

[1]Computing Security, Rochester Institute of Technology

September 6, 2024

Abstract -
The Internet of Things has allowed for widespread integration of technology in the physical world. As we near Industry 4.0, that connectivity will only continue to grow. This multiplies the consequences of IoT device failures. A shortcoming of many IoT devices is the write durability of their non-volatile storage, which is commonly flash memory. An attacker can exploit this limited durability by repeatedly writing to the device, which can physically destroy it. While various approaches to this problem have been proposed, prior work focuses on software level mitigations. In the event of a full software compromise, these controls are bypassed. This paper addresses that problem by proposing the use of a hardware module. This module directly limits the rate of writes to flash memory. Analysis of our approach reveals that we can considerably extend the lifetime of devices under attack while minimizing additional hardware cost.

1 Introduction

The Internet of Things (IoT) is a general term for networked devices. IoT devices span across all sectors, and include mobile phones, security cameras, medical devices, industrial control systems, radios [1], and more. The prevalence of IoT devices is on the rise, particularly in urban areas, as they're utilized in the pursuit of the Smart City vision [2].

IoT devices range in complexity but can be abstracted to a sensory microcontroller. Microcontrollers process sensor data for programmed output [3]. Microcontrollers are rarely modular and typically mandate whole, rather than piecewise, replacement. As such, IoT devices' lifespan is the sum of its components' predicted life [4].

As with all devices on the internet, IoT devices can be exposed to remote attackers operating across the network. Such devices provide a convenient attack vector while being laden with user data [5]. In the case of smart home devices, that data could be online account information, payment information, or private conversations. In the case of the Mirai botnet, millions of IoT devices were leveraged to perform Distributed Denial of Service (DoS) attacks [6]. However, IoT devices themselves can fall victim to Denial of Service attacks. Such attacks can have devastating consequences for medical or industrial uses.

In this paper, we focus on addressing a specific form of DoS attack performed against IoT devices. Attackers can perform high volume writes to burn out flash memory, which renders the device useless as a result. The continued use of IoT medical devices makes these attacks particularly impactful to people[7] as well as causing significant business impact should devices constantly need to be replaced.

Attacks like these can be difficult to prevent. Software modules exist to extend the memory lifetime of IoT devices [4] without accounting for malicious writing. Security oriented mitigations use a software-based design [8], but a total compromise of the device's operating system is a compromise of those tactics.

We aim to mitigate the effects of non-volatile memory-based DoS attacks on IoT devices via architectural support. Identifying and preventing high volume writes guards the life and function of the flash memory component.

We were successful in designing a platform-agnostic hardware module that effectively rate-limits writes to non-volatile flash memory. Using a simple fixed window rate-limiting algorithm, our solution establishes a minimum lifetime and has a negligible impact on hardware cost. Our solution also remains effective even in the event of a full software compromise.

2 Background

The *Internet of Things* is admittedly a broad term with many interpretations. A large portion of "IoT" takes advantage of passive RFID technology to store and transmit data between devices. IoT also encompasses sensor networks communicating via the traditional IP stack. In general, all IoT devices can be defined as "a world-wide network of interconnected objects

uniquely addressable, based on standard communication protocols" [9]. IoT devices can also range in complexity; some devices are simply used for identification of objects, while others run complex applications to analyze and respond to data in real-time. This paper will be focused on these more complex devices, but further research could be done to analyze similar security issues related to passive IoT devices.

Devices typically follow a *Service-Oriented Architecture* (SOA). While an application runs on the device and handles the processing and communication of data, a middleware solution sits beneath the application to interpret and categorize sensor data. Hardware sensors collect the initial data [9]. In practice they take the form of a microcontroller device, embedded into a system, that integrates with some object/sensor(s). Foundational microcontroller components include a CPU, memory (discussed below), a clock, and peripheral sensors & indicator mechanisms [3]. Applications are written to memory and run directly on the device. Due to their embedded nature, microcontroller design optimizes for cost and physical space. Replacing individual components is a complex process. As such, it's imperative that all components making up the microcontroller have comparable unified lifespans [3].

Microcontrollers feature volatile and non-volatile memory [3]. Volatile memory, also known as random-access memory (RAM), stores temporary data during run-time and is designed to handle a large number of read/write operations. Volatile memory is cleared when the device is powered off. In order to store long term data such as the application binary itself, non-volatile memory storing data after power-off is required [3]. This memory is optimized for long term reliability rather than high volume read/write operations. While there are many forms of non-volatile memory, a popular type is NAND flash memory for its compact and reliable design [10].

The DoS attack presented in this paper is immediately feasible in IoT microcontrollers. Through application layer compromise, an attacker can write to the device's flash memory [4][8]. The attacker could then issue a sustained high number of writes to artificially decrease the lifespan of the device.

Due to the embedded nature of microcontrollers and IoT devices, replacing end of life flash components is not practical. Components are proprietary and soldered to the board. Once flash memory has been burned out due to excessive writes, the device is considered "bricked" [8].

3 Related Work

At the time of writing, few publications directly address non-volatile memory-based denial of service attacks. No current design implements an architecture based control. Background research primary served to identify true NAND lifetime compared to vendors' claims.

3.1 Extending Lifetime of Device Memory

Threat actors' primary intention through non volatile memory DoS attacks is premature device failure. Loss of service via flash failure can also result from natural degradation independent of malfeasance.

Several papers focus on extending the lifetime of non-volatile memory within IoT devices to prevent device failure. Primary mechanisms include utilizing I/O schedulers [11], partial memory operations [12], file system enhancements and optimizations [13][14][15][16], wear-leveling techniques [17][18], and software controlled caches [19][4]. All outlined mitigations exist on a software level. I/O scheduling, partial memory optimizations, and file system amendments only reorganize the destination of malicious writes. Wear leveling and I/O schedulers delay but do not prevent the impact of high volume writes. Wear leveling only diffuses single bit or single address targeted attacks. Partial memory operations' benefit is significantly decreased on longer used hardware, making it insufficient for our use case. The proposed ELF file system is a new platform, rather than a platform agnostic addition.

Aras et al.[4] propose an embedded software solution with similar aims which is given the name MicroVault. This implant interfaces between the sensor application and physical memory. MicroVault uses adaptive techniques reducing the write volume to non-volatile memory. It has the ability to identify certain actions being taken by the application and use that data when determining what techniques to use. Counters are handled via grey coding, and further data is pushed to EEPROM random access memory rather then flash memory. The wear rate is then calculated. Given a configured cache interval, this is leveraged as it is more longevity oriented than allowing the device to blindly cache. The final step is to enable error correction if data can be written to the specified memory.

While MicroVault successfully and effectively extends device lifetime via numerous methods, it distributes load rather than limiting the volume of non-volatile memory writes.

3.2 DoS Attacks Against Mobile Phones

Zhang et al.[8] analyzed a non-volatile memory-based DoS attack via smartphone applications. In this paper, a modified Linux kernel monitoring I/O usage on the device and rate limiting techniques given a malicious volume of writes was proposed as a solution. Traditional usage was found not to exceed 5% of the 160miB/s sequential write output. Given a conservatively estimated lifetime, researchers allocated a variable amount of writes per day to ensure longevity. A rate-limiting approach protects from volumetric attacks that would otherwise quickly "brick" the device. This implementation assumes device kernel integrity. While the proposed solution in this paper is an effective rate-limiting strategy, it fails in the event of a full software compromise.

3.3 Commercially Available Products

The hardware vendor Micron does provide some hardware security options for non-volatile memory, however their feature set does not consider DoS attacks. Their "Hardware Write Protect" feature mandates valid voltage and grounding to interact with blocks of memory. Their optional LOCK pins disable writing for configurable ranges or entire NAND arrays. Given the non-modular design of microcontrollers, this is aligned with the low-level protection in our work. "Volatile" and "non Volatile Block Locking" prevents 'unexpected writes'. Ultimately, the scope of these protections is only offered to certain Micron products, a minor section of the Internet of Things devices in use. Micron advertises this product as configurable on power on, whereas an architecture based approach provides constant, non-propietary protection.

4 Methodology

As outlined in related work, prior solutions to non-volatile memory-based DoS have been fully software-layer approaches. Assuming full software compromise, an attacker would have the ability to modify any software based controls. Thus, we chose to implement our solution as a hardware module in order to mitigate against the possibility of software bypass and ensure the integrity of our solution.

A secondary benefit of a hardware module is its low resource impact. Abstracted implementations require hosting and integration with firmware. This includes using non-volatile memory to store the program, temporary memory for variables and logic, and logical connections to hardware and the device's installed software.

In this research we measure device lifetime as the time until single-bit errors begin. Single-bit errors are when a memory cell fails to erase, and demarcate first stage of flash memory degradation. Texas Instruments' MSP430 memory classification sheet states flash memory on the device can encounter an average of 10^5 read/write cycles before single-bit errors occur [20]. This data point is used later to calculate estimated device lifetimes.

4.1 Testing Environment

Our hardware module is written in Verilog, and integrated with the 16-bit OpenMSP430 microcontroller. This particular microcontroller was chosen for its open source access. The OpenMSP430 is modeled after the Texas Instruments MSP430, making it a well documented design. Our work can be adapted to other microcontroller architectures.

In order to effectively evaluate our design, we compiled and ran a mock application on the OpenMSP430 device to simulate malicious writes to flash. The program writes 2-byte chunks of data into non-volatile storage 100 times. These numbers are arbitrary, but meant to simulate a compromised application whose goal is to perform memory writes at an excessive rate. This application was written in C and compiled using simulation tools provided with the OpenMSP430 source code.

The device, with our integrated design, was synthesized and simulated using Vivado Machine Learning Integrated Development Environment v2018.3. Vivado is the industry standard for hardware design, as cited in [4]. Vivado synthesis defines resource requirement estimates using Lookup Tables *LUTs* and Flip Flops *FFs*. Vivado simulation provided numerical data for estimating device lifetime with and without the design, as well as time estimations for legitimate writes to non-volatile memory like software updates.

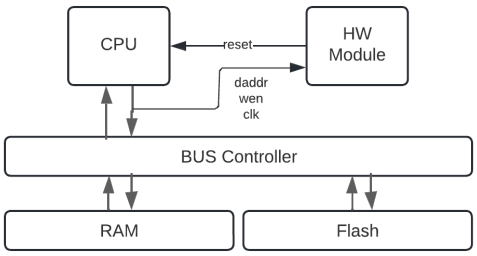

Figure 1: The hardware module will probe *wen*, *daddr*, and *clk* to approve a write or send a RESET signal.

4.2 Hardware Module Interaction

In this research, we focused on microcontrollers that communicate with memory via a bus controller. Our module determines time and destination of writes as a sidecar probing signals from the CPU. The signals that are monitored are write enable *wen*, data address *daddr*, and clock *clk*. The clock is used to synchronize our module, and write enable informs the module when a write action is performed.

Only non-volatile flash memory data addresses will be written to, per the data address signals. Non-volatile storage starts at address 0xE000 in our OpenMSP430 synthesis. This is the targeted address defined in the malicious application.

4.3 Rate Limiting Algorithm

Our design adapts the simple fixed window algorithm. The algorithm uses three main parameters: flash memory address range, window size and write threshold. The window size is given as a function of time in clock cycles and write threshold is given as an integer. Together, these two variables determine the overall rate of writes per microsecond, *writes/µs*. Each processed write is checked against flash address range to determine whether or not is within the range. The hardware module then increments the counter for the current window and will discard the write action if the counter exceeds the threshold for flash writes. This discard is accomplished by sending a RESET signal to the CPU. When the window size is reached, the counter resets back to 0 and a new window cycle begins.

This algorithm has the advantage of a low hardware cost, but foregoes some of the advantages of other rate limiting algorithms with its strict boundaries. In a period of low writes, if a burst of writes occur near the end of the *n-1* window and another burst of writes occurs at the beginning of the *nth* window, the operation incurs unnecessary loss. Alternative algorithms such as Sliding Log and Sliding Window address this issue by issuing weights to the number of writes to diffuse bursts, however both introduce a greater hardware cost. Our work focuses on overall device writes regardless of volume spikes.

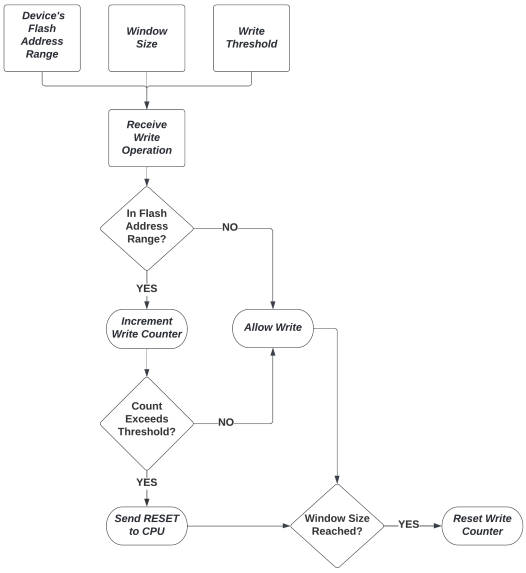

Figure 2: This diagram depicts the general flow of the hardware module.

5 Analysis & Evaluation

This section discusses the testing environment used to evaluate our proposed design, as well as the evaluation results and considerations to make when implementing this design.

5.1 Baseline Analysis

Without our implemented hardware design, the malicious application is able to write 200 bytes worth of data in the span of 729.53µs; this can be extrapolated to a rate of 0.274 writes per µs (writes/µs). As discussed previously, the MSP430 flash memory characteristics document states that flash memory on the device can normally withstand 10^5 write cycles before single-bit failures begin to appear [20]. Extrapolating this data, it would take approximately 0.364 seconds for the malicious application to start causing single-bit errors. From a benevolent standpoint; the time it takes to write to memory will impact legitimate behaviour such as the time needed to deliver software updates. With the current observed write-rate, it would take 0.292s in order to overwrite all 8KB of program memory on the device.

It should be noted that the provided OpenMSP430 test bed uses a clock that runs at 100MHz. Physical implementations of the MSP430, as well as other microcontroller devices, may run with different clock frequencies; this will alter device lifetime estimates. Our data, and extrapolations made with such data, are unlikely to match real-world observations. The behaviour patterns observed, however, should be applicable to real-world implementations.

5.2 Hardware Costs

Our hardware design, when synthesized in Vivado, had negligible increases to resource usage. With our solution implemented, Look Up Table (LUT) usage went from 12,170 to 12,196. This is a less than 1% change. The number of Flip Flops (FF) increased from 1,581 to 1,613, approximately a 2% increase.

Table 1: Δ FF & LUT Usage

	FF	LUT
Without Module	1581	12170
With Module	1613	12196
Increase	≈ 2%	≈ 0.2%

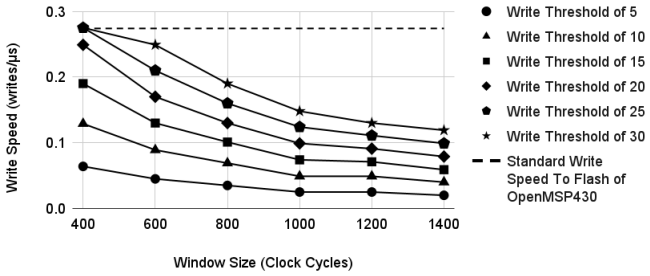

Figure 3: Δ write speed varying window size with a fixed value for the write threshold.

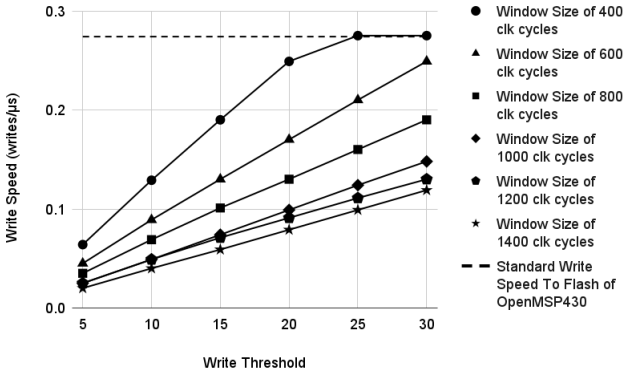

Figure 4: Δ write speed varying write threshold with a fixed value for the window size.

5.3 Choosing Parameter Values

Several parameters contribute to the aggressiveness of the rate limiting. Namely, the write threshold and window size parameters control the maximum number of writes allowed per window and the size of the time window in terms of clock cycles. As noted in section 4.2, these parameters determine the overall rate of writes per microsecond (writes/µs). This write speed directly effects the increase to device lifetime, but also negatively effects the speed of legitimate writes. An example of this is that software updates will be proportionately slower based on the increase in flash lifetime. To this end, our solution ultimately leaves the decision up to the manufacturer to determine what write speed is optimal to balance the aforementioned trade offs.

In order to gain a better insight into exactly how each parameter affects the write speed, we chose a wide range of argument values and ran simulations with our malicious DoS application to determine the resulting write rates. In these simulations, we fixed one parameter and independently varied the other to generate Figures 3 & 4.

This data informs an inverse relationship between window size and write speed. The converse is also supported. Both figures include the baseline write speed calculated in Section 5.2 without the hardware module.

To investigate write speed's impact extending the lifetime guarantee of a device, we provide an example simulation performed with a write threshold of 10 and a window size of 1000 cycles. These values are arbitrary. A real-world implementation would define parameters based in the minimum lifetime need of the device being configured. With these boundaries, the malicious application was only able to write 50 bytes worth of data over the span of 1011.93µs, which evaluates to a write-rate of 0.049 writes per µs. In this scenario, the lifetime of the device before single-bit failures being to appear is increased to 2.024 seconds; this is a 5.5 time increase in device lifetime under constant siege.

The cost of extended lifetime using this hardware design is a negative impact to the time needed to perform software updates. An 8KB device would require 1.619s in order to be completely overwritten; 5.5 times longer than without the hardware.

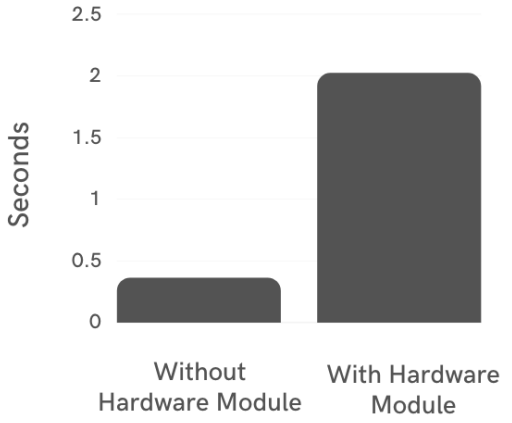

Figure 5: Device lifetime increased greatly with hardware module implemented.

5.4 Considerations

It should be reiterated that these results are highly dependent on the configuration parameters. Even small tweaks can result in large changes of device lifetime and write-efficiency. These factors will always be inverse with each other. When implementing this design in a real-world environment, considerations should be made regarding the desired minimum lifetime of the device and the size of typical software updates. A proposed intelligent design to configuring our design would be to modify the parameters such that small software updates are not hindered by rate-limiting, but that larger windows of memory writing will be seen as malicious and trigger a reset. This approach provides balance between functionality and protection.

6 Conclusion & Future Work

In conclusion, our fixed-window method successfully extends the lifetime of a device against non-volatile memory-based DoS attacks. It is capable of guaranteeing a minimum lifetime for a device with respect to its write durability. Additionally, the hardware cost of the device is not significantly impacted. The exact impact of our solution is determined by the parameter values configured at manufacture time. Optimal values provide the minimum write rate while meeting the legitimate use-cases of a device.

Our design is rooted in simplicity. Throughout the course of our research we carefully attended to the hardware footprint and additional cost of manufacture incurred by our module. In doing so, certain features were excluded from the scope of our research, and doors were opened for future research.

6.1 Preventative Actions

Our design utilizes a simple and inexpensive method to prevent excessive writes to flash memory. We elected to simply reset the device by setting the reset signal to high. This enables our proof of concept to function effectively and prevent damage to the embedded device while leaving the door open for further research.

An example of a more elegant solution could be to simply ignore writes that exceed the write threshold. This would allow the embedded device to continue functioning while preventing damage to the flash memory. This requires software to be written with additional safety checks in place, and research needs to be done to determine the additional hardware footprint of an approach like this.

6.2 Other Non-Volatile Memory

Our research solely focused on investigating flash memory as the attack target, but other types of non-volatile storage could still be affected by this attack. Further work could investigate the use of our module and varying our parameters on different hardware configurations, such as NOR non-volatile memory and FRAM non-volatile memory.

6.3 Numerical Analysis of Module Configuration

In our analysis, we explored parameter variation's effect on write speed. To do so, we simply fixed either window size or changed the write threshold (figures 3 and 4). However, it would be ideal to solve for a general equation that can be used for parameter tuning. This could potentially be solved with bivariate interpolation/extrapolation and by constructing a bilinear polynomial. This equation may reduce the time for manufacturers to determine the ideal parameters for their device use-case which would presumably lower business cost.

6.4 Dynamic Configuration

This module mandates that the rate-limit parameters are hard-coded at manufacture time. It could be worth investigating solutions that allow for more dynamic configurations. One idea is to store the parameter values in software and then encrypt them with keys stored in a secure hardware vault on the device. Examples of such a vault include Hardware Security Modules (HSM) or Trusted Platform Modules (TPM). Our approach is static as such security features are not guaranteed on all devices.

6.5 Codebase

The source code to our hardware module can be found on Github here.

References

[1] Atwell C. Everything You Need to Know about LoRa and the IoT; 2019. Accessed: 2022-02-08. https://www.designnews.com/electronics-test/everything-you-need-know-about-lora-and-iot.

[2] Zanella A, Bui N, Castellani A, Vangelista L, Zorzi M. Internet of Things for Smart Cities. IEEE Internet of Things Journal. 2014;1(1):22-32.

[3] Davies JH. MSP430 Microcontroller Basics. Elsevier; 2008.

[4] Aras E, Ammar M, Yang F, Joosen W, Hughes D. MicroVault: Reliable Storage Unit for IoT Devices. In: 2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS); 2020. p. 132-40.

[5] Heise Online. No-Name-Smart-Home: Security flaw allows easy firmware upload; 2019.

[6] Mirai Botnet Linked to Dyn DNS DDoS Attacks; 2018. Available from: https://www.flashpoint-intel.com/blog/cybercrime/mirai-botnet-linked-dyn-dns-ddos-attacks/.

[7] Yang G, Jiang M, Ouyang W, Ji G, Xie H, Rahmani AM, et al. IoT-Based Remote Pain Monitoring System: From Device to Cloud Platform. IEEE Journal of Biomedical and Health Informatics. 2018;22(6):1711-9.

[8] Zhang T, Zuck A, Porter DE, Tsafrir D. Apps Can Quickly Destroy Your Mobile's Flash: Why They Don't, and How to Keep It That Way. In: Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services. MobiSys '19. New York, NY, USA: Association for Computing Machinery; 2019. p. 207–221. Available from: https://doi.org/10.1145/3307334.3326108.

[9] Luigi Atzori GM Antonio Iera. The Internet of Things: A survey. In: Computer Networks. vol. Volume 54, Issue 15; 2010. p. 2787-805. Available from: https://www.sciencedirect.com/science/article/pii/S1389128610001568.

[10] Salajegheh M, Wang Y, Fu K, Jiang AA, Learned-Miller E. Exploiting {Half-Wits}: Smarter Storage for {Low-Power} Devices. In: 9th USENIX Conference on File and Storage Technologies (FAST 11); 2011. .

[11] Cui J, Wu W, Nie S, Huang J, Hu Z, Zou N, et al. VIOS: A Variation-Aware I/O Scheduler for Flash-Based Storage Systems. In: Gao GR, Qian D, Gao X, Chapman B, Chen W, editors. Network and Parallel Computing. Cham: Springer International Publishing; 2016. p. 3-16.

[12] Poudel P, Milenković A. Saving Time and Energy Using Partial Flash Memory Operations in Low-Power Microcontrollers. In: 2020 21st International Symposium on Quality Electronic Design (ISQED); 2020. p. 183-9.

[13] Chen D, Ye C, Ding C. Write Locality and Optimization for Persistent Memory. In: Proceedings of the Second International Symposium on Memory Systems. MEMSYS '16. New York, NY, USA: Association for Computing Machinery; 2016. p. 77–87. Available from: https://doi.org/10.1145/2989081.2989119.

[14] Dai H, Neufeld M, Han R. ELF: An Efficient Log-Structured Flash File System for Micro Sensor Nodes. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems. SenSys '04. New York, NY, USA: Association for Computing Machinery; 2004. p. 176–187. Available from: https://doi.org/10.1145/1031495.1031516.

[15] Tsiftes N, Dunkels A, He Z, Voigt T. Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks. IPSN '09. USA: IEEE Computer Society; 2009. p. 349–360.

[16] Gal E, Toledo S. A Transactional Flash File System for Microcontrollers. In: 2005 USENIX Annual Technical Conference (USENIX ATC 05). Anaheim, CA: USENIX Association; 2005. Available from: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/transactional-flash-file-system-microcontrollers.

[17] Woo YJ, Kim JS. Diversifying wear index for MLC NAND flash memory to extend the lifetime of SSDs. In: 2013 Proceedings of

the International Conference on Embedded Software (EMSOFT); 2013. p. 1-10.

[18] Hu XY, Haas R, Eleftheriou E. Container Marking: Combining Data Placement, Garbage Collection and Wear Levelling for Flash; 2011. p. 237-47.

[19] Chiou D, Jain P, Rudolph L, Devadas S. Application-Specific Memory Management for Embedded Systems Using Software-Controlled Caches. In: Proceedings of the 37th Annual Design Automation Conference. DAC '00. New York, NY, USA: Association for Computing Machinery; 2000. p. 416–419. Available from: https://doi.org/10.1145/337292.337523.

[20] Instruments T. MSP430 Flash Memory Characteristics; 2018. Available from: https://www.ti.com/lit/an/slaa334b/slaa334b.pdf?ts=1649446986575.