MicroVault: Reliable Storage Unit for IoT Devices

Emekcan Aras
imec-DistriNet, Computer Science
KU Leuven, Belgium
emekcan.aras@cs.kuleuven.be

Mahmoud Ammar
imec-DistriNet, Computer Science
KU Leuven, Belgium
mahmoud.ammar@cs.kuleuven.be

Fan Yang
imec-DistriNet, Computer Science
KU Leuven, Belgium
fan.yang@cs.kuleuven.be

Wouter Joosen
imec-DistriNet, Computer Science
KU Leuven, Belgium
wouter.joosen@cs.kuleuven.be

Danny Hughes
imec-DistriNet, Computer Science
KU Leuven, Belgium
danny.hughes@cs.kuleuven.be

***Abstract*—The Internet of Things (IoT) is being deployed at large scale in a wide range of long-life applications. Examples range from Industry 4.0 to smart lighting systems. These applications have diverse requirements of non-volatile storage. However, the flash memory that is used in today's IoT devices offers limited write endurance and must therefore be carefully managed if applications are to deliver on their promises of multi-year lifetimes. Managing the health of flash memory is difficult for application developers, as it requires in-depth hardware and software knowledge, which often needs to the problem being neglected. While various techniques have been proposed to preserve the health of flash memory, prior work tends to focus on a single hardware platform and data type. Furthermore, prior work does not provide lifetime *guarantees*. This paper tackles this problem by proposing MicroVault, a simple and unified interface for reliable non-volatile data storage on resource-constrained IoT devices. MicroVault enforces developer-specified lifetime guarantees through a range of lifetime extension techniques, which are adaptively applied based upon the needs of the application. Evaluation shows that MicroVault dramatically extends the lifetime of flash memory while minimising overhead.**

***Index Terms*—Embedded Software, Data Storage, Memory Management, Reliability, Internet of Things**

I. Introduction

Contemporary IoT devices are used in a wide and growing range of application domains. With the rise of Industry 4.0, smart buildings and smart cities, the IoT is becoming a vital part of infrastructure such as public spaces, electricity grids, hospitals and fleets of vehicles [1]. In contrast to Wireless Sensor Networks (WSN) or early IoT applications, contemporary IoT applications place increasing demands on non-volatile storage. The vast majority of contemporary IoT devices are based on off-the-shelf Micro Controller Units (MCUs) that are optimised for low power and low cost. These devices may be deployed in mains or battery powered systems. In the latter case, two decades of IoT and WSN research have enabling multi-year battery lifetimes [2]. This naturally leads us to focus on other challenges that may limit the lifetime of IoT devices, such as flash memory endurance.

Contemporary IoT devices offer non-volatile memory based on flash technology. Unlike volatile memory such as RAM, the physical properties of flash memory limit its write endurance. State of the art IoT devices deliver flash endurance of 10,000 to 100,000 cycles. This poses a problem for long-lived IoT applications that require reliable non-volatile storage throughout their lifetime. Furthermore, attackers can take advantage of this limitation to launch cyber-attacks that cause the physical destruction of these devices. Due the highly integrated design of IoT devices, it is likely infeasible for damaged parts to be replaced.

This paper tackles the issue of reliable multi-year storage for the IoT by proposing MicroVault, a software-based reliable storage unit for resource-constrained IoT devices. MicroVault enables the developer to establish a lower-bound lifetime guarantee for non-volatile IoT storage. MicroVault achieves this by adaptively applying four well-known techniques: (i) gray-coding, (ii) error-correction, (iii) micro-caching, and (iv) wear-levelling. Evaluation shows that MicroVault provides a dramatic improvement on the Flash memory lifetime of examined MCUs while incurring minimal overhead in terms of memory throughput. The contributions of MicroVault are three-fold:

1) The first non-volatile storage system[1] for IoT devices that enforces life-time guarantees for non-volatile storage across a range of devices and data types.
2) Adaptation techniques that optimise efficiency by selecting the most appropriate flash health techniques based upon evolving application requirements.
3) A thorough evaluation of the performance of MicroVault while serving various application requirements.

The remainder of the paper is organized as follows. Section II provides background on IoT storage requirements and flash memory health. Section III then introduces the design of MicroVault. Evaluation results are then reported in Section IV. Section V analyses related work. Finally, Section VI concludes and gives directions for future work.

II. Background

Internet of Things (IoT) devices are increasingly being deployed in long-life applications. On the one hand, this

[1] https://github.com/arasem/MicroVault

includes devices that are embedded in long-lived Industry 4.0 equipment [3]–[5] On the other hand, it includes a growing range of battery powered IoT devices that are capable of offering operational lifetimes of over 10 years on a single battery [6], [7].

The Micro Controller Units (MCUs) used in conventional IoT devices were typically designed for single-purpose applications with predictable low data rate storage requirements. However, this is increasingly at odds with rapid IoT product development and evolution. Ensuring that flash storage meets application lifetime goals in the face of heterogeneous hardware and evolving application requirements is complex for the application developer and therefore often neglected.

The remainder of this section explores the scope of this problem through two application case-studies, which are presented in Section II-A. Section II-B then provides background on flash memory technologies, the reliability of which is discussed in Section II-C. Based upon this analysis, we highlight requirements for MicroVault in Section II-D.

A. Application Case Studies

In this section, we illustrate the importance of reliable long term storage for the IoT through two case studies in the field of smart lighting and Low-Power Wide Area Network (LPWAN).

1) Physically destructive attack on smart lighting: To evaluate the memory lifetime of typical smart home IoT devices, we selected a WiFi-based smart bulb that is manufactured by Tuya [2]. At the time of writing, Tuya connects over 100 million smart devices and handle more than 80 billion device requests per day from 220 countries [8]. The Tuya light bulb uses an ESP8285 System on Chip (SoC), which embeds a WiFi radio, a Tensilica L106 32-bit MCU along with a 1MB of Flash memory [9]. Tuya devices can be interacted with through an open API[3]. We do not have access to the firmware source code and we therefore treat the device as a black box.

The weak security of smart lighting devices has been highlighted in prior work [10]. We followed a direct approach to capture the ID of the device and its local secret key during the commissioning phase (details of the attack is detailed in the Appendix B). Leveraging the captured information and the standard Tuya API, we created a script to repeatedly make slight changes to the brightness and colour temperature of the bulb. Even when the bulb was in use, these changes were too small to be noticed. The device saves each of these changes to EEPROM, so that the bulb will power on in the same state.

After 5.8 million parameter changes over 25 days, we were able to physically destroy the EEPROM of this device, which caused complete system failure. To validate our finding, we conducted the same experiment on an ESP8285 development board, where we also managed to destroy the flash after 641 thousand cycles.

2) Premature Device Failure in LoRaWAN: LoRaWAN is a widely used Low-Power Wide Area Network (LPWAN) protocol for the IoT. It is available through more than 100 operators in over 100 countries world wide [11], making it an important protocol in the IoT landscape.

LoRaWAN uses AES-128 in CTR mode to provide confidentiality, authenticity and integrity for transmitted messages. For correct operation, AES128-CTR requires that devices store an incrementing frame counter in non-volatile memory to prevent replay attacks when devices restart [12]. The rate at which LoRa devices may send messages is dependent upon the Spreading Factor (SF) of the device and regional duty cycle regulations. This limits a device to sending between 30 messages (SF12, EU band G2, 0.1% duty cycle) and 7500 messages per hour (SF7, EU band G3, 10% duty cycle). If devices update their frame counters upon every transmission, their EEPROM will be destroyed in between 12 hours and 4 months. This is particularly unacceptable as LoRa has gained significant traction for supporting communication with Smart City and Industry 4.0 infrastructure.

We performed experiments to validate the durability of non-volatile memories on three different widely used LoRaWAN development boards; Lora32u4 (BsFrance), RN2483 (Microchip) and STM32 LoRa Discovery (STM). The empirical validation showed that LoRa32u4 and STM32 LoRa board do not take any countermeasures and the non-volatile memory of these devices were broken after only a few hours of experiments. On the other hand, MicroChip RN2483 module utilizes countermeasures which kept it running reliably long after the write endurance of its host microcontroller. The details of empirical validation can be found in the Appendix A.

B. Flash Memory

Flash memory is the most ubiquitous form of non-volatile storage available in conventional MCUs. According to industry convention, EEPROM refers to byte-wise writable flash, while 'Flash' typically refers to block-wise writable memory. To avoid further confusion we use the same convention throughout the rest of the paper.

Flash memory is typically constructed from NAND gates, while EEPROM is built from NOR gates. The architectural differences between EERPOM and Flash memory affects the area density and functionality of the resulting storage medium. A Flash memory cell of the same capacity is physically smaller than equivalent EEPROM, however, accessing a single cell of Flash is not possible due to the NAND gate architecture [13]. EEPROM memory is therefore preferred for flexible small-scale storage, while Flash delivers high density storage at a lower cost.

C. Memory Reliability

Due to their physical properties, every Flash and EEPROM memory has a limited number of write-cycles or *'endurance'*. This limit varies widely across chips. However, in general, Flash memory tends to wear out faster than EEPROM due to the use of block-wise operations, which impact all cells in the block regardless of how much data has been written. Table I highlights the durability of various non-volatile memories that

[2]https://en.tuya.com/

[3]https://github.com/codetheweb/tuyapi

TABLE (I) Memory and Size Comparison [14]

MCU Family	Model	EERPOM Size	EEPROM Endurance (Erase/Write Cycles)	Flash Size	Flash Endurance (Erase/Write Cycles)
Atmel 8-bit Avr	AT32U4	1 KB	100000	32 KB	10000
NXP 16-bit Hc12	MC68HC705	4 KB	30000	-	-
CYPRESS Psoc1	CY8C29X	-	-	32 KB	50000
INFENION 32-bit TriCore	TC212L	64 KB	125000	512 KB	125000
Microchip PIC16	PIC16F1574	128 B	100000	14 KB	10000
Silicon Labs C8051	C8051F300	-	-	8 KB	100000
STM32-F0	STM32F0x0	-	-	16-256 KB	10000
TI MSP430	MSP430G2X	-	-	2 KB	10000

are used in popular IoT platforms. As can be seen from Table I, the endurance of EEPROM and the ratio of available Flash memory to EEPROM varies widely across MCU families.

D. Requirements Analysis

Based upon the analysis conducted above, we identify five key requirements for the reliable long term storage of IoT data.

1) *Guaranteed lifetime:* Rather than employing best-effort reliability techniques, we advocate that IoT storage systems should support the explicit specification and enforcement of lifetime requirements.
2) *Manage hardware heterogeneity:* MCUs offer various mixes of EEPROM and/or flash with different endurance and capacity. A generic approach is required to deliver reliable non-volatile storage on IoT devices and thereby reduce complexity of developing long life applications.
3) *Manage software variability:* IoT applications have different storage requirements in terms of both data rates and variable types. Reliable long term storage must efficiently deal with different data types.
4) *Intelligent Caching* : While Flash and EEPROM have limited endurance, RAM does not. Exploiting this property through the use of caching has the potential to dramatically extend the lifetime of embedded devices.
5) *Minimal Overhead:* Software for IoT applications must execute efficiently within the memory, computation and energy limitations of IETF Class-1 devices [15] (10kB RAM and 100kB Flash) as these are the most common devices currently deployed in the field.

In Section III, we introduce the design of MicroVault, a system that is designed to address these requirements.

III. MicroVault

In this section, we present the design of MicroVault, a practical and reliable software storage unit for IoT devices. The software architecture and APIs of MicroVault are developed around the following principles:

- *Unified interface to long term storage* that operates with different software and hardware.
- *Eliminate low-level complexity* by providing a reliable API independent from hardware and applications.
- *Intelligent memory trade-offs* should be made between different memory mediums to maximise the life-time of non-volatile memory.

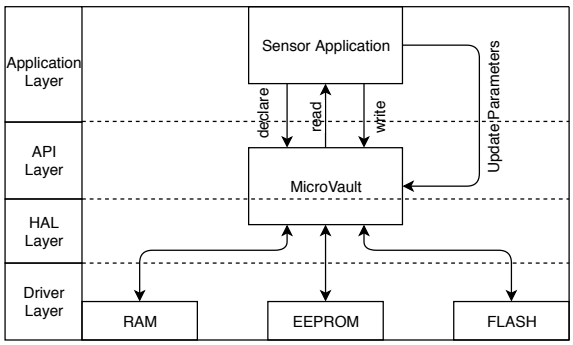


Fig. (1) MicroVault Software architecture.

A. Software Architecture

Figure 1 shows the MicroVault architecture which consists of three layers: low-level drivers, a hardware abstraction layer and the MicroVault interface layer which exposes a standard API to higher level software.

- The **low level Driver layer** provides an interface to the different types of memory offered by the IoT device. The current version of MicroVault supports three different memory types; (i) RAM, (ii) EEPROM and (iii) Flash. The software architecture utilizes a low-level memory library with a simple read/write functionality, which makes MicroVault compatible with many types of MCU.
- The **hardware abstraction layer** implements adaptable memory access functionality. This layer serves as a bridge between the MicroVault API and the low-level drivers, configuring low level parameters and selecting the most suitable technique to extend the memory life-time based on application requirements.
- The **MicroVault API** provides a unified interface between the application and long term storage. To use MicroVault, an application first declares a variable using the API and then configure lifetime and data parameters(wear-levelling size, read/write rate per day, data consistency requirements). Based upon these parameters, MicroVault establishes a lower-bound on guaranteed life-time and then informs the application of it. The API is compatible with many embedded operating systems, including: FreeRTOS[4] or TinyOS [16], as well as frameworks such as Arduino[5] or PlatformIO[6]. It may also be included with bare-metal software.

B. Adaptive Approach

MicroVault is built around an adaptive algorithm which combines different techniques and applies the best combination of flash health techniques. The selected combination of techniques depends upon the available memory types and their size as well as the consistency and reliability requirements of the application, as defined through the MicroVault API.

[4]https://www.freertos.org
[5]https://www.arduino.cc
[6]https://platformio.org

Algorithm 1: MicroVault Adaptive Algorithm

```
Result: Allocate Data in MicroVault
begin
    if Data is Counter then
        Activate Gray Coding;
    end
    if EEPROM is avaliable then
        Prepare EEPROM;
    else
        Prepare FLASH;
    end
    Calculate wear rate;
    // Based on required lifetime, consistency
       requirements and write rate
    if Micro-Caching is on then
        Prepare Micro-Cache;
        Adjust wear_levelling_size;
    end
    Check available space on prepared memory;
    if available memory ≥ wear_levelling_size then
        Set wear levelling frames;
    else
        Set maximum wear levelling frames possible;
        Activate Error Correction Algorithm;
    end
    Notify the application of guaranteed life time;
end
```

Algorithm 1 provides a high-level overview of the adaptive algorithm behind MicroVault. Once a variable is declared through the API, MicroVault calculates memory wear rates based on the flash endurance limit, which is provided by the vendor and the application requirements, which are given as an argument through the API upon declaration of the variable. MicroVault then applies the best combination of flash health techniques to maximise memory life-time. If the guaranteed life-time cannot be met, Error Correction Algorithm is activated as a last resort.

C. Flash Health Techniques

The techniques below are developed and optimized to achieve low-energy overhead and latency while protecting long-term data storage against corruption.

- **Micro-caching** : Caching is a well-known technique commonly used in mainstream computing to improve performance. High-end processors use independent caches, that are realised as on-chip memory or implemented on RAM, where the data is stored temporarily [17]. MicroVault implements micro-caching to limit memory wear, while keeping memory footprint minimal. Micro-caching is an adaptive version of regular caching. While in regular caching, data is updated during every operation, the micro-cache updates the non-volatile memory only periodically, based upon an interval configured by the application.
- **Wear-levelling** is a common technique used in both high-end and embedded computers to mitigate the limited write cycles of flash. Wear levelling distributes data across the memory to extend the time until any single location reaches its endurance limit [18]. MicroVault applies an adaptable wear-levelling policy that may be applied across regions of Flash or EEPROM.
- **Reducing bit-flips**: The storage of incrementing counters is common in IoT applications. However a single counter increment may flip multiple bits. MicroVault reduces unnecessary bit flips by 50% for incrementing counters using Gray coding. This technique generates a coded value by performing an XOR operation between the value and a right shifted version of it. This ensures that the Least Significant Bit (LSB) changes every second count, the next bit every forth and so on [19].
- **Error Correction**: Flash and EEPROM memory failure is usually limited and random, which enables its detection and correction using error-correction algorithms. MicroVault implements an optimized version of Reed-Solomon (RS) [20] error correction to increase the life-time of the memory cells. The RS algorithm is used as a last resort to increase lifetime in cases where the aforementioned techniques cannot achieve a requested life-time.

D. Implementation

MicroVault is written in C and uses standard vendor drivers wherever possible. Currently, MicroVault has been implemented for three MCU architectures: AVR, STM32 and MSP430. Software was compiled with AVR-GCC, Arm Embedded GCC and msp430-gcc tool-chains. Our prototype implementation of MicroVault has been tested with the Arduino and Eclipse Development environments and executed on various development boards.

IV. Evaluation

This section evaluates MicroVault against the application requirements described in Section II-D. We begin in IV-A by describing our experimental methodology. Section IV-B then evaluates the impact of each technique on memory capacity. Section IV-C investigates performance implications. Section IV-D sketches low power characteristics of MicroVault. Section IV-E we present an integrated evaluation using all techniques. Finally, Section IV-F discusses the implications of these findings.

A. Experimental Methodology

All experiments were conducted using a representative IoT Micro Controller Unit (MCU), the ATMega328. This MCU offers an 8 bit 16MHz processor, 2kB of RAM, 32kB of Flash and 1kB of EEPROM. Flash memory may be read in single bytes, but must be written in sectors or *pages* of 128B, while EEPROM provides single byte read/write access. Flash has an

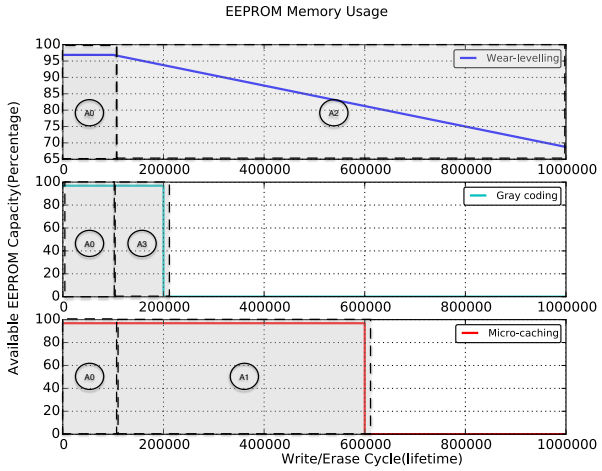


Fig. (2) Impact of Discrete Lifetime Extension Techniques on Memory Capacity

endurance of 10K cycles, while EEPROM has an endurance of 100K. In the case of micro-caching the memory checkpoint parameter was set to 5 (i.e. variables are transferred from volatile to non-volatile memory after 5 updates). The power consumption is calculated using average execution time, input voltage (3.3 V) and flash/eeprom programming current.

B. Impact of Individual Techniques on Memory Capacity

We evaluate the effectiveness of wear-leveling, gray coding and micro-caching on memory lifetime. In this evaluation, we focus on EEPROM. The results for Flash follow an identical pattern with a 10x reduction in lifetime.

Figure 2 shows the impact of using each technique to store 32B of sequentially increasing counter data in EEPROM for a guaranteed number of read/write cycles. Areas of the graph are annotated from *A0* to *A3*, each of which is explained below:

A0 - Standard Lifetime: The A0 zone shows the standard lifetime of EEPROM. Without additional techniques, 100% of the EEPROM is available for 100K write cycles.

A1 - MicroCaching provides a linear extension in memory lifetime, as variables are only written to EEPROM after 5 updates. As expected, this leads to a 5x increase in memory lifetime to 500K write cycles, at the expense of a bounded inconsistency in stored data.

A2 - Wear-Levelling extends lifetime by spreading read-/write cycles across a greater area of memory. As can be seen in A2, the relationship between lifetime extension and memory used is linear. In the case of a 32B variable, this enables a 10x extension of lifetime at the cost of 32% capacity.

A3 - Gray Coding extends the lifetime of flash memory by using an alternative binary representation to minimise bit-flips for incrementing counters. As A3 shows, this extends the lifetime of flash memory by 2x, with no impact upon capacity. However this is only possible for counter storage.

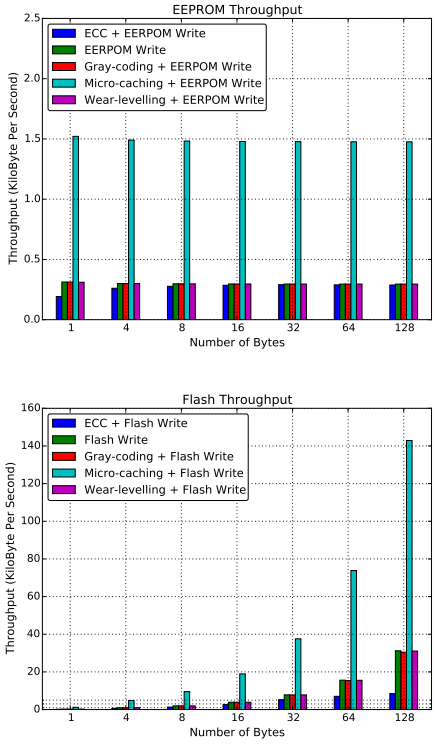


Fig. (3) Performance Overhead over different memories

As Error Correction Codes (ECC) provide a non-deterministic extension of memory lifetime rather than guarantees, we do not evaluate their performance in this section.

C. Impact of Individual Techniques on Memory Performance

We evaluate the execution overhead of each lifetime extension technique when writing data blocks of 1B to 128B as shown in Figure 3. As the aim of MicroVault is to provide a unified interface to reliable long term storage, it allows single byte read/write access to both Flash and EEPROM.

As expected, the error correction algorithm has the highest execution time and therefore lowest throughput for both memory types. This leads to an overhead of 79% to 300% for Flash and 2% to 89% for EEPROM when compared to standard write operations. The impact of gray-coding and wear levelling is lower with a worst case overhead of 0.4% for EEPROM and 0.07% for Flash respectively. Micro-caching actually improves storage performance by up to 100x depending on the data size as writing to RAM is significantly faster than writing to Flash or EEPROM. The throughput of Flash memory increases with data size as the MCU must write 128B pages in all cases.

D. Impact of Individual Techniques on Power Consumption

Figure 4 shows the energy consumption per byte during write operations. As seen in the figure, the energy profile does not change on EEPROM depending on the size whereas a drastic decrease is observed in the power consumption of flash memory. In addition, the limited overhead of both gray coding and wear-levelling keeps the energy overhead low

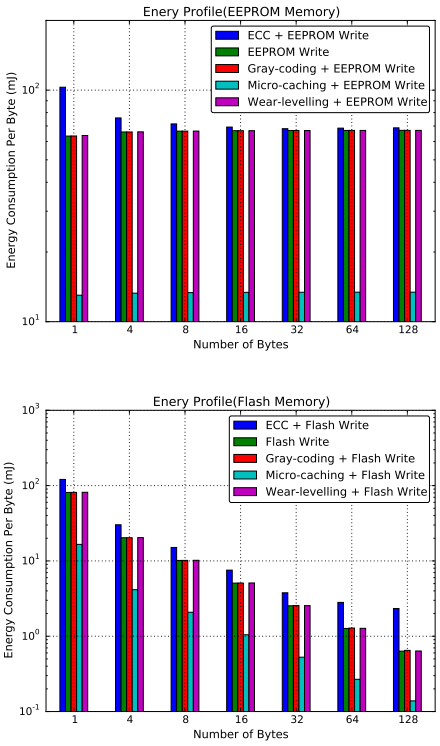


Fig. (4) Power consumption over different memories

which increases the energy consumption by only 0.44% and 0.077% respectively.

E. Integrated Case-Study

In this section, we evaluate the overall impact of MicroVault for three representative use-cases. In all cases, the sensor node collects 32B of data which is stored in non-volatile memory 10 times per hour using MicroVault. The application requires 10 years of reliable operation, which equates to a memory endurance of 876K writes. All use-cases have a consistency requirement that data cannot be more than *two readings out of date*. As shown in Figure 5, the default 100K write endurance of EEPROM is inadequate to support our application use-cases. However, the lifetime extension techniques of MicroVault enable the platform to meet its lifetime goals in all cases. We review the implications of using Micro-Vault for each case below:

1) *Data logging:* arbitrary data such as sensor readings is periodically stored in non-volatile memory. As this data is not a counter, Gray coding provides no benefits and is not used. However, micro-caching (A1) extends memory lifetime by 2 years with no capacity penalty. The remainder of the lifetime requirement is achieved using wear-levelling (A2), which delivers a 8 year extension at the cost of 15% of EEPROM capacity.
2) *Counter storage:* An incrementing counter is logged in non-volatile memory as required for AES128-CTR encryption in LoRaWAN. As before, micro-caching (A1) delivers a 2 year extension of battery life with no loss of capacity. As the variable is a counter, MicroVault also applies Gray coding (A3) which delivers an additional 2 years of memory lifetime. The remaining 6 years are delivered using wear-leveling (A2), leading to a capacity reduction of 9.3% of EEPROM capacity.
3) *Data logging with limited storage:* This use case is the same as (1), however, the memory that may be devoted to lifetime extension is limited to 8% or 82 bytes of EEPROM capacity. In this case, it is not possible for MicroVault to guarantee the required 10 year lifetime. MicroVault first applies micro-caching (A1), followed by wear levelling (A2) until the capacity limit is reached at 6 years. After this point, MicroVault applies error correcting codes, (A4) to achieve an additional best-effort lifetime extension.

TABLE (II) Configuration set by MicroVault for Different Use-cases

Use-Case	Micro-Caching Frame Size	Wear-Leveling Frame Size	Gray Coding	Error Correction Algorithm	Guaranteed Life-Time
Data Storage	2	5	Off	Off	135 months
Counter Storage	2	3	On	Off	165.6 months
Data with Limited Memory	2	3	Off	On	82.8 months

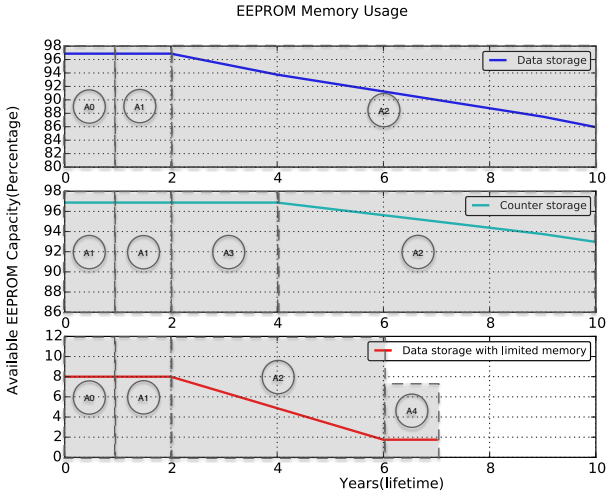
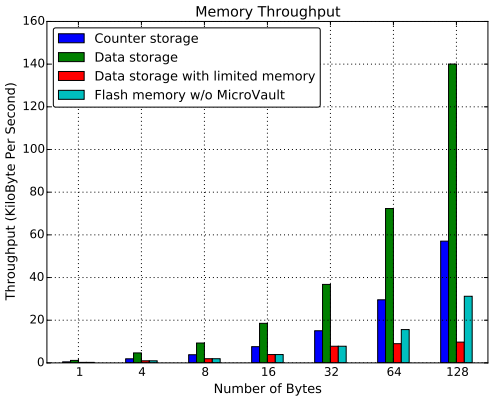
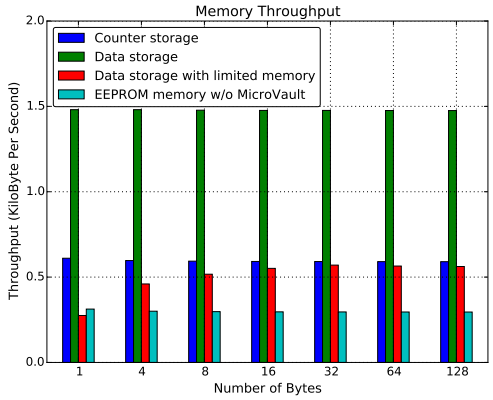


Fig. (5) Impact of Integrated Micro-Vault Techniques on Memory Capacity

In addition, we perform a memory throughput assessment for the integrated techniques embodied by MicroVault. The settings for this evaluation are shown in Table II. The performance of these configurations is shown in Figure 6. As all use-cases apply micro-caching in RAM, MicroVault often increase memory throughput. In the data-storage with limited memory case, MicroVault achieves better throughput in Flash when writing smaller data sizes of 1 to 32B, compared to native flash memory throughpout in contrast to EEPROM where writing larger data achieves better performance against EEPROM throughput. Using a larger wear-levelling frame does not affect MicroVault performance over sequential operations since the

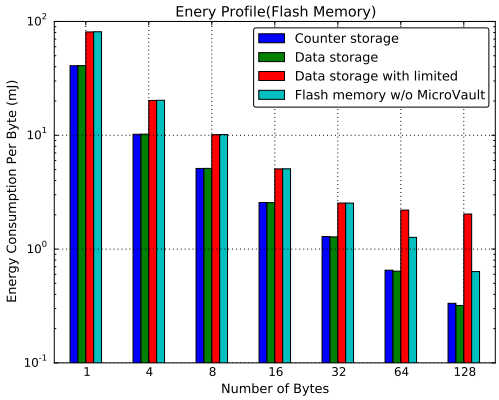


(a) Flash Memory Throughput

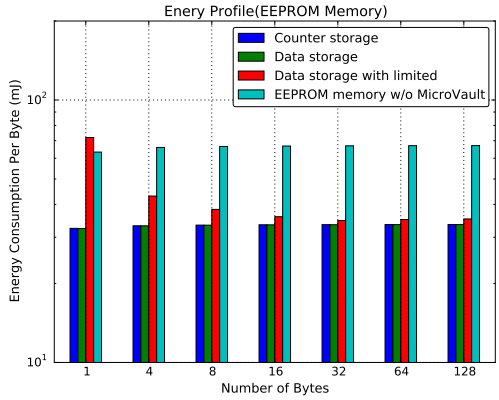


(b) EEPROM Memory Throughput

Fig. (6) Performance Overhead in Difference Use-cases



(a) Flash Memory Energy Consumption



(b) EEPROM Memory Energy Consumption

Fig. (7) Energy Overhead in Different Use-cases

execution time to find a next frame does not change depending on the number of frames.

To conclude our evaluation, we performed an additional evaluation to sketch the energy consumption of the MicroVault to validate low-power characteristics. The energy profile of MicroVault can be seen at Figure 7. As expected, energy consumption depends on native memory overhead. The energy cosumed by ECC has a significant impact on flash memory for both EEPROM and flash. However, implementing micro-caching lowers energy overhead in comparison to native memory operations.

F. Discussion

Our evaluation results show that MicroVault meets the requirements outlined in Section II-D, delivering a guaranteed lifetime for long-term IoT storage. The generic MicroVault interface adapts its data storage techniques based upon the type of variables that it is required to store. Furthermore, the overhead of MicroVault is minimal and well within the capabilities of IETF Class-1 devices.

Considering prior approaches to IoT flash health such as Coffee [21] and Matchbox [22], MicroVault requires only half of the memory footprint of Coffee and one sixth the footprint of Matchbox. Furthermore, MicroVault consumes 178B of static RAM without micro-caching, which is the same value as Coffee and one fifth the RAM required by Coffee.

V. Related Work

A. Embedded File Systems

Several file systems have been proposed for embedded devices that consider flash memory health. ELF [23] was one of the first Flash-based file systems to target resource-constrained devices. It supports common sensor file operations while keeping the power consumption minimal.

TFFS [24] is another file system for embedded devices. It supports concurrent transactions and atomic operations as well as crash recovery. While TFFS has a limited Flash footprint, it uses significantly more RAM compared to other file systems and thus it is not suitable for the IETF Class-1 devices that are targeted by MicroVault.

Tsiftes et. al. [21] argue for a generic, high-speed, Flash-based file system that supports a wide range of sensor devices. This vision is realised in Coffee. Unlike other Flash-based file systems, Coffee has a constant RAM footprint per file and avoids excess flash wear by levelling it across all pages. How-

ever, the use of wear levelling without any other techniques limits memory lifetime gains in comparison to MicroVault.

B. Mainstream Flash Management Systems

Memory wear has been studied widely on mainstream computers. With the development of solid state drivers (SSD), the reliability and durability of Flash has become a major issue. In contrast to the wear-levelling algorithms which use erase count as a wear levelling index, Woo et.al [25] argue for a more sophisticated wear-levelling algorithm that takes into account more diverse properties of Flash memory such as error patterns within a Flash block and bit error counts. Although their proposed algorithm improves Flash life-time compared to existing techniques, it cannot be applied on resource-constrained embedded devices.

Flash memory is also widely used in consumer mobile devices. As stated in [26], the Flash memory of mobile devices may wear out rapidly under heavy use. As with embedded devices, it is not possible to replace memory upon failure. Moreover, a malicious application or malware can make a smartphone unbootable in few weeks. The authors propose to tackle this through a rate-limiting algorithm that restricts the usage of problematic applications, thereby protecting the smartphone from wearing-out its Flash.

VI. Conclusion and Future Work

Contemporary IoT applications are requiring longer operational lifetimes while demanding higher data-rates and reliability. This paper tackled this challenge by introducing MicroVault, an adaptive software-based reliable storage unit that dramatically extends the lifetime of non-volatile memory for IETF Class-1 devices. MicroVault limits memory wear by adapting and optimising four well-know techniques: (i) micro-caching, (ii) wear-leveling, (iii) gray-coding, and (iii) Reed-Solomon-based error correction. Our Evaluation results show that MicroVault is able to prolong the operational life of non-volatile memory far beyond the manufacturers specification, while preserving a relatively low overhead in terms of memory footprint and power consumption.

Our initial evaluation of MicroVault demonstrates the need for a reliable software storage unit for embedded devices and motivate us to extend the work further. Although, MicroVault is designed to be compatible with popular types of memory technologies(RAM, Flash and EEPROM), the library may also be extended to achieve compatibility with emerging memory technologies such as FRAM or NVRAM. Using different type of memory will expose different trade-off between cost and performance and thereby require the development of new algorithms and techniques for MicroVault. In addition, the MicroVault library can be extended to support more conventional mass memory mediums such as SD cards.

References

[1] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things journal*, vol. 1, no. 1, pp. 22–32, 2014.

[2] Y.-J. Kim, H. S. Bhamra, J. Joseph, and P. P. Irazoqui, "An ultra-low-power rf energy-harvesting transceiver for multiple-node sensor application," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 11, pp. 1028–1032, 2015.

[3] Intel, "Intel's fog reference design overview." Online, 2017.

[4] A. A. Zhilenkov, D. D. Gilyazov, I. I. Matveev, and Y. V. Krishtal, "Power line communication in iot-systems," in *2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pp. 242–245, IEEE, 2017.

[5] Siemens, "Simatic iot2020." Online, 2017.

[6] F. Yang, N. Matthys, R. Bachiller, S. Michiels, W. Joosen, and D. Hughes, "μpnp: Plug and play peripherals for the internet of things," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, Association for Computing Machinery, 2015.

[7] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Proceedings of the 4th international symposium on Information processing in sensor networks*, p. 48, IEEE Press, 2005.

[8] Tuya, "Tuya smart - the world's leading iot platform." Online.

[9] Espressif, "Esp8285 datasheet." Online.

[10] H. Online, "No-name-smart-home: Security flaw allows easy firmware upload." Online.

[11] C. Atwell, "Everything you need to know about lora and the iot." Online.

[12] I. Butun, N. Pereira, and M. Gidlund, "Security risk analysis of lorawan and future directions," *Future Internet*, vol. 11, no. 1, p. 3, 2019.

[13] G. Schatzberger, F. P. Leisenberger, P. Sarson, and A. Wiesner, "High efficient low cost eeprom screening method in combination with an area optimized byte replacement strategy which enables high reliability eeproms," in *2018 IEEE 36th VLSI Test Symposium (VTS)*, pp. 1–6, IEEE, 2018.

[14] S. Evanczuk, "The most-popular mcus ever." Online.

[15] C. Bormann, M. Ersue, and A. Keranen, "Terminology for constrained-node networks," *Internet Engineering Task Force (IETF): Fremont, CA, USA*, pp. 2070–1721, 2014.

[16] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, "Tinyos: An operating system for sensor networks," in *Ambient intelligence*, pp. 115–148, Springer, 2005.

[17] D. Chiou, P. Jain, L. Rudolph, and S. Devadas, "Application-specific memory management for embedded systems using software-controlled caches," in *Proceedings of the 37th Annual Design Automation Conference*, pp. 416–419, ACM, 2000.

[18] X.-Y. Hu, R. Haas, and E. Evangelos, "Container marking: Combining data placement, garbage collection and wear levelling for flash," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 237–247, IEEE, 2011.

[19] K. Zeger and A. Gersho, "Pseudo-gray coding," *IEEE Transactions on communications*, vol. 38, no. 12, pp. 2147–2158, 1990.

[20] S. B. Wicker and V. K. Bhargava, *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[21] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the coffee file system," in *2009 International Conference on Information Processing in Sensor Networks*, pp. 349–360, IEEE, 2009.

[22] D. Gay, "Design of matchbox, the simple filing system for motes," 2003.

[23] H. Dai, M. Neufeld, and R. Han, "Elf: An efficient log-structured flash file system for micro sensor nodes," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 176–187, ACM, 2004.

[24] E. Gal and S. Toledo, "A transactional flash file system for micro-controllers.," in *USENIX Annual Technical Conference, General Track*, pp. 89–104, 2005.

[25] Y.-J. Woo and J.-S. Kim, "Diversifying wear index for mlc nand flash memory to extend the lifetime of ssds," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, p. 6, IEEE Press, 2013.

[26] T. Zhang, A. Zuck, D. E. Porter, and D. Tsafrir, "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*, pp. 207–221, ACM, 2019.

APPENDIX A
EMPIRICAL EXPERIMENTS

In order to investigate flash memory health, a set of empirical experiments have been performed. The software used in the experiments contains a simple erase/write task to maximize the throughput of the operation. As previously mentioned, widely used microcontrollers from AVR family, AT32U4, and STM32 family, STM32F0x0, were used in these experiments, the results of which can be seen in Figure 8. What stands out from the both graphs is the significant number of failures that occur after the guaranteed life-time by vendors. STM32 Flash claims 10k reliable erase/write cycles and fails after 100k cycles while AT32U4 EEPROM claims 100k reliable cycles and it fails after 600k. We observed single bit failures after exceeding endurance limits yet the number of failures were limited and are therefore not visible in both figures. Moreover, failure starts with small number of bits and slowly increases.

The first failure usually can be observed after 6 hours on the EEPROM and 2 hours on the flash. This empirical validation proved that the memory wear remains a challenge on conventional MCUs and that high-data rate applications can easily reach the limit for reliable cycles within the lifetime of the device.

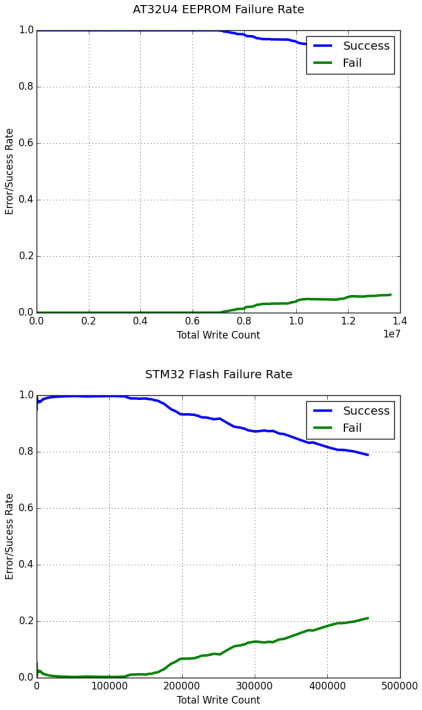


Fig. (8) Emprical Experiments on Flash and EEPROM

APPENDIX B
DETAILS OF SMART LIGHTING ATTACK

In general, to register a new WiFi-based smart bulb through the Tuya companion app on the smartphone, the smart bulb should be in the *smartconfig* mode of operation, wherein it is not connected to a WiFi network and listens for UDP packets on a predetermined port (in this case 6669). Taking into account that the communication between any WiFi-based Tuya device and its companion app is cloud-based, the companion app must start by asking the cloud for a token and secret key. Afterwards, the app sets up a UDP packet with a specific structure which contains the token (that is valid for a few minutes), the secret key (which will be the local key of the device), the SSID and password of the WiFi network that the device should connect to, and a Cyclic Redundancy Check (CRC) to check the integrity of the packet once delivered. It then broadcasts this packet on the pre-defined port after encoding the data in the length of a UDP packet. The smart bulb receives this packet, validates it by checking the CRC, extracts the data, and connects to the WiFi network. This network must have an Internet connectivity, so the smart bulb can register with Tuya cloud and verifies the token received. If this step ends up successfully, the cloud refreshes the companion app by adding this smart bulb to its list of registered devices. By then, both the companion app and the smart bulb can communicate securely through the cloud, by encrypting messages through the shared secret key.

We were able to obtain the secret key, smart device ID and other sensitive without physical attack simply by setting another device in the transmission range of the smartphone that hosts the companion app to passively eavesdrop on the communication and read packets (a passive man-in-the-middle-attack). Alternative techniques, such as compromising the insecure companion app, have been already reported to leak these credentials.

All Tuya devices can be interacted with through an open API as discussed previously. Leveraging the credentials that e obtained (the device Id and its secret key), we have configured a Raspberry PI to communicate secretly with the Internet-connected smart bulb. Our script loops indefinitely, slightly changing the brightness and colour temperature of the smart bulb, so that even if it was turned on, such change will never be detected by the smart home users. This change requires reading and writing to EEPROM. Recalling that all communications with Tuya smart devices are cloud-based, we had to insert a delay of 1 second after reading/writing the Flash memory for 50 times to avoid cloud-side DDOS prevention policies.