Light Flash Write for Efficient Firmware Update on Energy-harvesting IoT Devices

Songran Liu[1], Mingsong Lv[1,2*], Wei Zhang[3], Xu Jiang[1], Chuancai Gu[4], Tao Yang[4], Wang Yi[1], and Nan Guan[5]

[1]Northeastern University, China
[2]The Hong Kong Polytechnic University, Hong Kong
[3]Quancheng Laboratory, Jinan, China
[4]Huawei Technologies Co., Ltd., China
[5]City University of Hong Kong, Hong Kong

***Abstract*—Firmware update is an essential service on Internet-of-Things (IoT) devices to fix vulnerabilities and add new functionalities. Firmware update is energy-consuming since it involves intensive flash erase/write operations. Nowadays, IoT devices are increasingly powered by energy harvesting. As the energy output of the harvesters on IoT devices is typically tiny and unstable, a firmware update will likely experience power failures during its progress and fail to complete. This paper presents an approach to increase the success rate of firmware update on energy-harvesting IoT devices. The main idea is to first conduct a lightweight flash write with reduced erase/write time (and thus less energy consumed) to quickly save the new firmware image to flash memory before a power failure occurs. To ensure a long data retention time, a reinforcement step follows to re-write the new firmware image on the flash with default erase/write configuration when the system is not busy and has free energy. Experiments conducted with different energy scenarios show that our approach can significantly increase the success rate and the efficiency of firmware update on energy-harvesting IoT devices.**

***Index Terms*—energy-harvesting, firmware update, IoT**

I. Introduction

IoT devices increasingly rely on energy harvesting to power themselves because they are deployed in complex working environments, and thus it is hard to charge or replace their batteries [1]–[4]. Without energy harvesting, the lifetime of IoT devices will be severely limited by the battery size. As the energy output of harvesters is typically tiny and unstable, an energy-harvesting IoT (EH-IoT) device typically executes intermittently: the device runs for a short while and depletes the energy storage; after that, the device has to shut down to collect energy and restarts after the energy storage is replenished. This unique nature brings critical challenges to tasks requiring much energy to execute. Such a task may frequently experience power failures before it completes. Many important system services will be severely affected by this problem [2], [5], among which firmware update is a crucial one [6].

EH-IoT devices are expected to serve for a long time once deployed [7]. The firmware of a device will often be updated to fix vulnerabilities and add new functions [6], [8]. Typically, a firmware server provides firmware update data. The firmware update process is to download update data from the server, generate a new firmware image in the main memory and save the image persistently to the nonvolatile memory on the device, in most cases a flash memory [9]. It is well-known that writing flash memory is a time and energy-consuming operation [10]. When the firmware update is performed on an EH-IoT device, writing flash memory quickly depletes the small energy storage and triggers a power failure. In this case, the firmware image stored in the main memory will be lost, and the update will fail. After the energy regains, the firmware update has to be restarted from fetching the firmware data from the server. The above problem may keep appearing in low energy conditions, and the firmware update may never succeed.

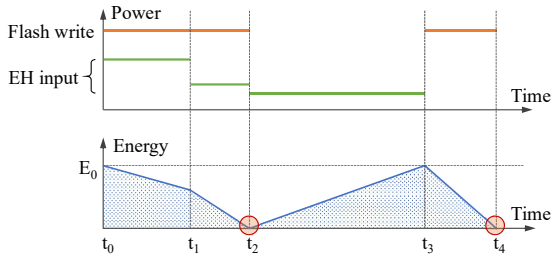


Fig. 1. An example of the firmware update problem

The above problem is demonstrated with the example in Fig. 1. The upper coordinate depicts the power consumption of writing flash memory and the power generated by the energy harvester over time. The lower coordinate shows the change in device energy. When the update starts, the device has initial energy of E_0. During $[t_0, t_2]$, although there is still power input, writing the flash is so energy-consuming that the energy storage is eventually depleted at time t_2. The device has to shut down, and the firmware update fails. From time t_2 to t_3, the energy storage is replenished. Then the device performs the firmware update for a second time. Unfortunately, from t_3 on, there is no energy input from the environment. The device energy is depleted at time t_4, causing a second firmware update failure.

In this work, we propose a solution to increase the success rate of firmware update on EH-IoT devices. The main idea is quickly to save the new firmware image to flash memory before the energy storage is depleted. Once done, there

This work is partially supported by Research Grants Council of Hong Kong (GRF 11208522, 15206221) and Natural Science Foundation of China (grant No. 61772123). Corresponding author: Mingsong Lv.

978-3-9819263-7-8/DATE23/© 2023 EDAA

will be no worries about losing the firmware image in the presence of power failures. This is achieved by reducing the time duration for erasing and writing flash segments, thus considerably reducing the energy needed. We call it "*light flash write (LFW)*". To ensure the firmware image written by LFW can retain for a time duration as long as that in the hardware specification, we conduct a reinforcement step after LFW, which re-writes the new firmware image with default flash erase/write configurations. As the firmware image is already on the nonvolatile flash memory, the reinforcement can be performed incrementally and intermittently and thus will not be impeded by power failures.

We conducted experiments to evaluate the effectiveness of the proposed approach and compared it with two baseline methods. Experimental results under different energy harvesting scenarios show that the proposed system considerably increases the success rate of firmware update and provides high efficiency for a firmware update in terms of the total time and energy used to complete the update.

II. Background on Flash Memory

Flash memory is one of the most widely used nonvolatile memories for IoT devices [9]. The basic storage element of flash memory is *cell*, a MOSFET transistor with a floating gate (shown in Fig. 2). A cell's charged and discharged states e used to represent logical bits "0" and "1", respectively. Multiple cells (typically several kilos) are embedded on the same *segment* of the flash memory. To modify a bit from 1 to 0, the cell will be charged. To modify a bit from 0 to 1, the cell should be discharged, but it is not allowed to discharge a cell independently. All the cells on the same segment must be discharged altogether, a process called *erase*. By erasing, some cells with bit 0 that should not be modified are discharged from 0 to 1. Then in a second step, such cells will be recharged, changing their bit value back to 0. A segment must be erased before it can be written on standard flash memory devices.

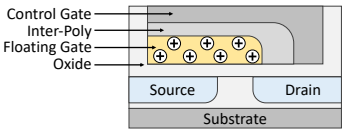


Fig. 2. The cell of a NOR flash [11]

There are two types of flash memories: NOR flash and NAND flash. NOR flash is comparably low-power, highly reliable, and capable of random access. These nice features make NOR flash suitable for low-power and resource-constraint Micro Controller Units (MCUs). NOR flash is widely adopted as an integral part of many MCU chips to store program code and data [12], [13]. Even though erasing and writing NOR flash is much more time and energy-consuming than reading flash and other operations on the device. For example, on an MSP430F5529 MCU [12], it typically takes around 27ms and 16ms to erase and write a 512-byte segment, respectively, and the power consumption can be 4-5 times larger than that of the CPU core. During regular service of an IoT device, the flash memory is used like a read-only memory most of the time. But in a firmware update, intensive flash erase/write operations require large energy consumption.

III. Overview of the Solution

The energy-consuming flash erase/write operations may cause a firmware update to deplete the device energy before it completes. As a result of power failure, the firmware image in the volatile main memory will be lost. After the device is recharged and restarted, a firmware update has to be re-initiated. In the worst case, power failures may continuously happen, and the firmware update may never finish.

In this work, we propose a technique called "*Light Flash Write (LFW)*" to solve the problem. The main idea is to quickly save the new firmware image to flash memory before the energy storage is depleted. Once done, there will be no risk of losing the firmware image. To achieve this, we reduce the time durations used to erase and write a flash segment and, thus, the energy consumed. There may be concerns that if we reduce the time for flash erase/write, the firmware image cannot retain on the flash memory for as long as in the hardware specification. To address this issue, we perform a reinforcement step after LFW. In this step, the segments of the new firmware will be read to the main memory and then written back to the same flash segments with default erase/write configurations. Note that the reinforcement step consumes much energy to finish, which means it may span multiple power cycles and experience power failures. But this is not a problem. As the firmware image is already on the nonvolatile flash memory, there is no worry about losing the new image during the reinforcement step. The reinforcement can be done incrementally and intermittently (in the granularity of one flash segment) when the system is not busy and has free energy so that it can make progress.

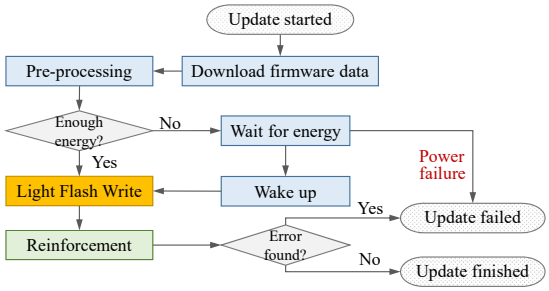


Fig. 3. The workflow of the proposed approach

The workflow of the proposed solution is given in Fig. 3. Once the firmware update data are downloaded to the device, a new firmware image is generated in the pre-possessing step[1], after which the number of segments to write is decided, and the energy required for LFW can be estimated. Then, we check if the energy currently in the energy storage is enough for LFW. If not, the device will sleep by entering a low-power mode and waiting for the energy storage to be replenished. Once the

[1]Typically, the firmware server only sends the differential data, and a new firmware image is generated based on the differential data and the old firmware image [14]. Pre-processing techniques are out of the scope of this paper. The QDiff approach [15] is applied in our work.

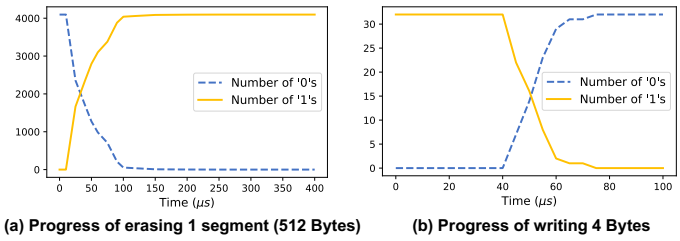
energy storage is fully charged, LFW is performed, and the new firmware image is written to flash memory with reduced erase/write time. After that, the reinforcement step is conducted, which may span across multiple power cycles. When reinforcement finishes, the firmware update is completed. In the following sections, we will detail the two main components of the proposed solution: LFW and reinforcement.

IV. Light Flash Write

The main idea of LFW is to reduce the energy consumed on flash erase and write operations by reducing the time used for flash erase/write. The details are provided in this section.

When an erase or write operation is executed, the flash device will enter a busy mode; when the device exits the busy mode until the flash erase/write operation finishes. The time spent in the busy mode is normally decided by the hardware specification. We use T_e and T_w to denote the time delays to erase and write a segment with default configurations. For example, T_e and T_w are around 27ms and 16ms on MSP430 MCUs. The erase and write times from the hardware specification are an over-design, as chip producers typically try to ensure a very long data retention time, such as 100 years [11].

In fact, it is possible to reduce the time used to erase/write the flash hardware and thus energy consumption [12], which was tested by us on an MSP430F5529 MCU, with the results shown in Fig. 4. To erase a flash segment of 4096 bits, all the cells stabilize at bit 1 after around $400\mu s$, compared to $27ms$ from the hardware specification. Since the maximum granularity of flash writes on MSP430 is 4 bytes, we tested the time used to write 32 bits from 1 to 0. All the cells stabilize at 0 after around $80\mu s$ (including bus preparation operations). In our test, it takes $7.8ms$ to write the whole flash segment[2], around half of the flash write time from the hardware specification ($16ms$).



(a) Progress of erasing 1 segment (512 Bytes) (b) Progress of writing 4 Bytes

Fig. 4. Exploration of reduced erase/write times

LFW exploits this opportunity to reduce the energy consumption of flash erase/write by reducing the operation time. In the implementation, we use the erase/write abortion operation on the MSP430 MCU to reduce the operation time. To erase a segment, a timer is first set with a target erase time T_e^l smaller than the default segment erase time T_e. Then flash erase is started. When the timer expires, the CPU will set the emergency exit bit EMEX of the flash memory controller. In this way, the erase operation is aborted, and the segment is erased with time T_e^l. The flash write operation is handled similarly in LFW. We use the above abortion technique to reduce the time to write a data block from T_{wb} to T_{wb}^l. To write a segment, a series of data blocks will be written consecutively, taking T_w^l time.

To decide the values of T_e^l and T_w^l, we conduct intensive experiments on the hardware and observe when data stabilize in a flash erase/write. The MSP430 MCUs provide the "marginal read mode" to check if the cells of a FLASH segment stabilize. In our design, each time the LFW is performed, we invoke a marginal read to test the stability of cell data. T_e^l and T_w^l are decided by taking the worst-case observed values added by a certain margin for high reliability. Now, to write a flash segment with LFW, the segment is first erased and then written with new data, with reduced operation time, using the above technique. This is shown by the yellow blocks of the workflow in Fig. 5.

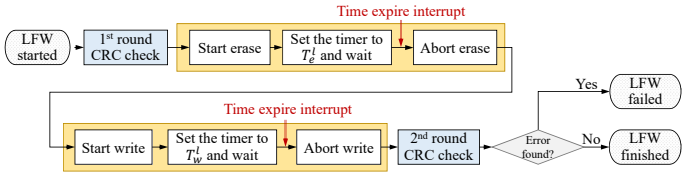


Fig. 5. The workflow of LFW (to erase and write one segment)

If the system designer still has concerns that data may be incorrectly written to the flash memory with reduced operation time, we offer a data integrity check, shown with the blue blocks in Fig. 5. To implement a data integrity check, we first compute the Cyclic Redundancy Check (CRC) on the new firmware image before LFW. After the new firmware image is written, each flash segment containing the new firmware image is read back, and a new round of CRC check is performed. If the checksums of the two rounds of CRC check are the same, the segment is known to have been correctly written by LFW. Specifically, we adopt the on-chip CRC hardware component (commonly available in most MCUs) to do CRC checks energy-efficiently. At the end of the data integrity check, the checksum for every segment will be written to flash memory (with default flash write configuration) since it will be reused in the reinforcement step. We dedicate a segment on flash memory to save the checksum. When there are multiple flash segments to write in an update, the yellow blocks in the workflow in Fig. 5 will be repeated, and the data integrity check can be performed all at once for all the segments.

V. Reinforcement

After LFW, we offer a reinforcement step to improve data stability. The main operation is to erase and write the segments again with default erase and write configurations when the device has enough energy. The workflow is shown in Fig. 6.

The main loop of the workflow is to read one segment from flash memory back to the main memory and then write back the data in the same location with default erase/write configurations. Since default erase/write operations consume considerable energy, the reinforcement for the whole firmware image may span several power cycles. So, at the beginning of each loop, the device will check if there is enough energy to reinforce at least one segment; otherwise, the device will shut

[2]To write a flash segment takes $7.8ms$ instead of $80\mu s \times (4096/32) = 10.24ms$ because some preparation bus operations can be omitted when writing 4-byte blocks continuously.

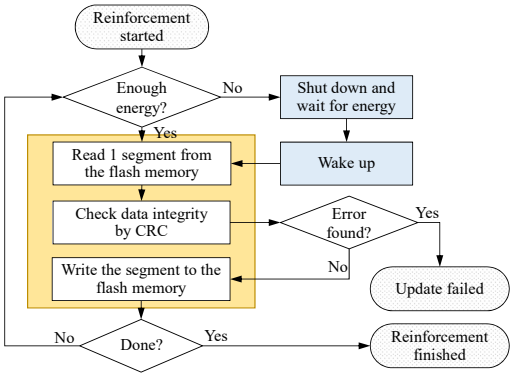


Fig. 6. The workflow of reinforcement

down and wait for the energy storage to be fully replenished. We adopted a common approach in energy-harvesting IoT systems to estimate the remaining energy: using an ADC on the MCU to read out the current output voltage of the capacitor and inferring the remaining energy with a profiled relationship between the voltage and the remaining energy. Note that a data integrity check exists between reading and writing a segment. This is to ensure that the data previously written to the flash memory with LFW are correctly retained so far. If an error does occur in extreme cases, the firmware update is deemed failed. The above steps loop until all the segments of the new firmware image are reinforced. In energy replenishment, we let the energy storage fully charge before continuing reinforcement to reduce the number of shut-downs and wake-ups.

In fact, the reinforcement step needs not be performed immediately after the LFW step since the data written with LFW will correctly persist on flash for quite some time. We have conducted intensive testing in which the firmware is only read under a common working scenario. We observed that after two months, there is no data loss on the flash segments written with LFW. So, there is a decent time window for the reinforcement to be done later (when energy input is abundant and the device is not busy). The reinforcement of multiple segments can be done in disjoint time windows, as the reinforcement of each segment can be performed independently.

VI. Experiments and Evaluation

In this section, we provide implementation details based on an MSP430 board. Then we provide the energy consumption in different steps of a firmware update profiled on the actual hardware. To evaluate the effectiveness of the proposed approach under different energy harvesting scenarios, we simulate several energy harvesting traces in our experiments. The proposed approach is compared with two baseline approaches regarding the success rate and efficiency of firmware update.

A. Implementation

We built an EH-IoT hardware platform (Fig. 7) and implemented LFW as part of the device bootloader. The platform comprises a TI MSP-EXP430F5529 board [16] as the main device and a TI LAUNCHXL-CC2640r2 board [17] to communicate with the firmware update server via Bluetooth.

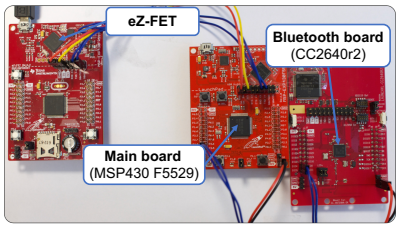


Fig. 7. The hardware platform

The two boards are connected with I^2C interface. The MSP-EXP430F5529 board has a TI MSP430F5529 MCU (clocked at 1MHz) and 128KB on-chip NOR flash. The segment size of the flash memory is 512 bytes, and the unit of flash write is 4 bytes. The BLE 5.0 protocol is adopted for wireless communication to download firmware update data.

Energy/Power profiling. We profiled the energy consumption for the firmware update steps on the hardware platform. An eZ-FET board [18] that implements TI's EenrgyTrace technology is adopted to measure energy consumption precisely. We measured each step 100 times and obtained the average energy consumption in Table I. According to the method in Sec. IV, T_e^l and T_{wb}^l is finally decided to be $500\mu S$ and $100\mu S$, respectively. Thus, to erase a flash segment, energy consumption drops from $137.2\mu J$ to $3.9\mu J$; to write a flash segment, energy consumption drops from $78.8\mu J$ to $20.8\mu J$.

TABLE I
Energy consumption in firmware update

Symbol	Description	Energy (μJ)
E_{adv}	BLE advertising for 1 time	52.6
E_{tran}	BLE transfer of 224 Bytes	29.1
E_{asm}	Assembling one segment, including reading old segments from flash memory and generating one new segment	0.5
E_{crc}	CRC check for one segment, including data movement with DMA	0.2
E_r	Reading one segment from flash memory with DMA	0.12
E_e	Erasing 1 segment with default configurations	137.2
E_{le}	Erasing 1 segment with LFW	3.9
E_w	Writing 1 segment (erased) with default configurations	78.8
E_{lw}	Writing 1 segment (erased) with LFW	20.8

With BLE 5.0, an advertising step must be performed to establish a connection between two communicating devices. During data transfer, BLE 5.0 transfers 224 bytes at a time. We also measured the power consumption when the hardware platform enters the low-power mode LPM4 that retains the main memory and switches off all clocks. The measured power consumption for the launchpad in LPM4 is $89\mu W$ (including those for the CPU core, SRAM data retention, and the hardware board). We denote this power consumption with P_{lp}.

B. Experimental Setup

Simulation of energy harvesting. We generate power traces to simulate different energy harvesting scenarios. Each power trace has multiple small time segments in which there can be transient energy input. The transient energy input power is in the range $[100\mu W, 300\mu W]$, and the average power of a whole power trace is in the range $[60\mu W, 120\mu W]$, which can exhibit both low and high energy input cases. For a power trace, we

TABLE II
BENCHMARK PROGRAMS USED FOR EVALUATION

Programs	Description	Data Size (bytes)	# Segments to Write
INK	App update for an intermittent OS – INK [19]	984	4
SEP	Security patch for betaflight [20]	1,008	7
DRV	Bug fixing for ESP32 XBee driver firmware [21]	2,336	8
LOS	Function upgrade to the RTOS LiteOS [22]	3,452	10
CNN	Weight update of a CNN, HAR-CNN [23]	6,148	13

TABLE III
THE NUMBER OF SUCCESSFUL FIRMWARE UPDATES FOR SOS AND LFW UNDER ALL POWER TRACES

Groups	INK			SEP			DRV			LOS			CNN		
	BL	LP	LFW	BL	LP	LFW	BL	LP	LFW	BL	LP	LFW	BL	LP	LFW
$S_{0,0}$	51	75	500	0	0	500	0	0	500	0	0	500	0	0	494
$S_{1,0}$	55	81	500	0	0	500	0	0	500	0	0	500	0	0	497
$S_{0,1}$	54	124	500	0	93	500	0	74	500	0	44	500	0	29	500
$S_{1,1}$	58	162	500	0	102	500	0	87	500	0	56	500	0	38	500
$S_{0,2}$	54	293	500	0	253	500	0	194	500	0	191	500	0	144	500
$S_{1,2}$	56	345	500	0	276	500	0	242	500	0	232	500	0	204	500

define the duty cycle as the percentage of time segments with energy input. The duty cycle ranges in $[50\%, 100\%]$, simulating both fluctuating and stable energy harvesting scenarios. Consider a solar-powered smart watch, when the person wearing the watch moves, the solar input can vary very often with the changing angle of the solar panel; when the person seats, the power input is stable. Each power trace simulates a time duration of 100 seconds, and we generated 3000 power traces to exhibit different energy harvesting scenarios.

Simulation of energy consumption. We wrote a simulator to simulate the energy consumption in a firmware update, using the profiled data listed above and the simulated power traces. Each time an update step is executed, the corresponding energy consumption is deducted from the energy storage; energy harvested from the environment will be added to the energy storage. In our evaluation, we assume an energy storage with a capacity of $400mF$ (denoted by E_0), a typical configuration of low-power EH-IoT devices.

Benchmark programs. We evaluate different approaches with a set of benchmarks listed in Table II that demonstrates typical firmware update scenarios, including RTOS update, security patch, bug fixing, neural network configuration update, etc. The size of the firmware update data and the corresponding number of segments to be modified are given in the table. Since update data may be distributed to different segments with different patterns, the number of segments to write is not proportional to the update data size.

Compared approaches. We evaluate and compare the proposed approach against two other approaches: 1) BL: the BL approach adopts normal flash erase and write and does nothing to cope with power failures during firmware updates; 2) LP: the LP approach lets the device enter a low-power mode to survive potential power failures during a firmware update. Specifically, in the LP approach, if the supply voltage of the energy storage drops to $2.3V$ (below which the flash memory will malfunction [12]), the device will enter the LPM4 mode. To compare the approaches, we obtain the total number of successful firmware updates regarding different energy harvesting scenarios. For LP and LFW, we also evaluate the efficiency of successful firmware updates in terms of the total time and energy consumed to complete a firmware update.

C. Results and Evaluation

Success rate. All benchmarks were evaluated under the 3000 generated power traces, and the numbers of successful firmware updates under the BL, LP, and LFW approaches are listed in Table III. The results are classified into groups w.r.t. energy harvesting power. Each group is denoted by $S_{i,j}$, where index $i = \{0, 1\}$ indicates duty cycle in range $[50\%, 70\%]$ and $[70\%, 100\%]$, and index $j = \{0, 1, 2\}$ indicates average power input in range $[60\mu W, 80\mu W]$, $[80\mu W, 100\mu W]$ and $[100\mu W, 120\mu W]$. The success rate is defined as the number of successful firmware updates out of all firmware updates, for each benchmark and each group.

First, the BL approach only has a small number of success cases for the smallest benchmark INK. This demonstrates that firmware image writing is too energy consuming and can hardly finish before a power failure occurs. Second, for the LP approach, an average success rate of 22.2% is observed, higher than BL, because allowing to enter the LPM4 low-power mode helps the device to survive some power failures. But the LP approach still has its problem. As LP adopts default flash erase/write, to write the whole firmware image, the device has to enter LPM4 many times. Thus, there is a high risk that the device will encounter power failure when it sleeps in low-power mode, leading to more firmware update failures.

Comparably, the proposed LFW approach is free from the above problems, and the firmware update succeeds in almost all experiments. Once the LFW step is finished, the firmware image is persistently saved on flash memory. The reinforcement step was performed incrementally and intermittently in the presence of power failures. The only extreme cases occur for group $S_{0,0}$ and $S_{1,0}$ and for benchmark CNN. In these settings, the energy harvesting power is too low and the firmware image sizes are too large, so the LFW step fails in a few cases.

Time & energy efficiency. To evaluate efficiency, we break down the total time and energy consumption for all benchmarks under the LP and the LFW approaches. The results are given in Table IV. For each benchmark, the numbers are averaged over all successful firmware updates. In the table, "Sleep Times" shows how many times the device slept under LP, and "Shut down times" shows how many times the device shut down under LFW. "Update" refers to the process in LP to write the firmware image to flash memory with default configurations. "Sleep" refers to the state that the device is in low-power mode under LP. "Shut down" refers to the state that the device shuts down under LFW. "LFW" and "REINF" refer to the LFW step and the reinforcement step of the LFW approach.

LFW reduced the time and energy consumption by 64% and 67%, respectively, compared to LP. A further look into LP shows that 99% of the total time and 72% of the total energy are spent in low-power mode, waiting for energy to be harvested. Under LFW, the device shuts down to wait for energy in the reinforcement step and thus wastes no energy. Even though LFW writes the firmware image twice (once in the LFW step, the other in the reinforcement step), the total energy

TABLE IV
TIME AND ENERGY CONSUMPTION FOR ALL BENCHMARKS AVERAGED OVER ALL SUCCESSFUL FIRMWARE UPDATES

Items			INK	SEP	DRV	LOS	CNN
LP	Sleep Times		0.31	1	1.44	2	3
	Time (ms)	Update	204	341	404	509	665
		Sleep	20,511	45,430	69,301	94,263	139,553
		Total	20,715	45,771	69,705	94,772	140,218
	Energy (μJ)	Update	1,080	1,730	2,127	2,713	3,545
		Sleep	1,836	4,066	6,202	8,437	12,490
		Total	2,916	5,796	8,329	11,150	16,035
LFW	Shut down times		1	1.02	2	2.91	4
	Time (ms)	LFW	81	104	132	163	207
		REINF	184	321	367	459	596
		Shut down	11,378	12,497	22,806	32,128	43,797
		Total	11,643	12,922	23,305	32,750	44,600
	Energy (μJ)	LFW	458	635	742	945	1,203
		REINF	867	1,517	1,733	2,167	2,817
		Shut down	0	0	0	0	0
		Total	1,325	2,152	2,475	3,112	4,020

consumption is still far less than that under LP. Comparing different benchmarks, the time and energy spent in low-power mode increase significantly with the number of segments to write. This is because the more segments to write, the more times the device will enter the low-power mode in the LP approach. This shows that entering low-power mode to survive power failures in firmware updates is not a good option.

From our experimental testing, data written with LFW can be retained on flash memory for a long time. So the reinforcement can be delayed and performed intermittently when the device has enough energy and no other work to do. In some application scenarios, if the data retention time under LFW is larger than the firmware update period, the reinforcement step can even be skipped, further reducing time and energy consumption.

VII. RELATED WORK

IoT devices are connected resource-constraint embedded devices and typically serve for years once deployed [7]. This nature requires the device's firmware to be updated often to fix vulnerabilities and add new functions [6]. Moreover, due to resource constraints, firmware updates must be performed resource-efficiently.

Recent research on firmware update on IoT devices mainly focuses on two issues: improving security [6], [14], [24], [25] and reducing firmware update data size. For secure firmware updates, cryptography algorithms are used to protect the firmware from tampering, and digital signatures are adopted to authorize the firmware images [26], [27]. The issue of reducing the size of firmware data was addressed in different dimensions [14], such as developing algorithms to find the common data between the new and old firmware images [28].

Recent research considered reducing the energy consumption of flash memory by reducing the supply voltage to the flash device [29]. In their work, the authors found that flash memory chips can tolerate voltage drops. Their technique can reduce the energy consumption of flash devices but only works on MLC NAND flash. Other similar works [10], [30] try to save flash energy consumption by reducing the flash operating voltage. These techniques cannot guarantee data correctness on flash memory written with low voltage.

VIII. CONCLUSION

We studied the problem of firmware update on energy-harvesting IoT devices. We presented a solution to avoid update failures due to power downs. The solution performs flash erase/write with reduced operation time to quickly save the new firmware image to the flash memory to keep firmware update away from the harm of power failures. A reinforcement step is followed to re-write the new firmware image with the default configuration to ensure a long data retention time. Experimental results demonstrate that the proposed approach significantly improves the success rate and efficiency of firmware update.

REFERENCES

[1] Xuecai Bao, Hao Liang, Yuan Liu, and Fenghui Zhang. A stochastic game approach for collaborative beamforming in sdn-based energy harvesting wireless sensor networks. *IEEE Internet of Things Journal*, 2019.

[2] Meng-Lin Ku, Wei Li, Yan Chen, and K. J. Ray Liu. Advances in energy harvesting communications: Past, present, and future challenges. *IEEE Communications Surveys and Tutorials*, 2016.

[3] Shwetak N. Patel and Joshua R. Smith. Powering pervasive computing systems. *IEEE Pervasive Comput.*, 2017.

[4] C. Xu, L. Yang, and P. Zhang. Practical backscatter communication systems for battery-free internet of things: A tutorial and survey of recent research. *IEEE Signal Processing Magazine*, 2018.

[5] S. Umesh and S. Mittal. A survey of techniques for intermittent computing. *Journal of Systems Architecture*, 2020.

[6] A. Kolehmainen. Secure firmware updates for iot: A survey. In *IEEE Smart Data*, 2018.

[7] Suk Kyu Lee, Mungyu Bae, and Hwangnam Kim. Future of IoT networks: A survey. *Applied Sciences*, 2017.

[8] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal, and Biplab Sikdar. A survey on iot security: application areas, security threats, and solution architectures. *IEEE Access*, 2019.

[9] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 2003.

[10] Mastooreh Salajegheh, Yue Wang, Kevin Fu, Anxiao Jiang, and Erik Learned-Miller. Exploiting Half-wits: Smarter storage for low-power devices. In *FAST*, 2011.

[11] Msp430 flash memory. https://www.ti.com/lit/an/slaa334b/slaa334b.pdf.

[12] Ti exp430f5529 mcu. https://www.ti.com/product/MSP430F5529.

[13] Stm32. https://www.st.com/en/microcontrollers-microprocessors.html.

[14] Konstantinos Arakadakis, Pavlos Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. Firmware over-the-air programming techniques for iot networks - a survey. *ACM Comput. Surv.*, 2021.

[15] N. B. Shafi, K. Ali, and H. S. Hassanein. No-reboot and zero-flash over-the-air programming for wireless sensor networks. In *SECON*, 2012.

[16] Exp430f5529 launchpad. https://www.ti.com/tool/MSP-EXP430F5529.

[17] Launchxl-cc2640r2. https://www.ti.com/tool/LAUNCHXL-CC2640R2.

[18] Msp430fr5994 board. https://www.ti.com/tool/MSP-EXP430FR5994.

[19] Kasım Sinan Yıldırım, Amjad Yousef Majid, and et al. Ink: Reactive kernel for tiny batteryless sensors. In *SenSys*, 2018.

[20] betaflight homepage. https://github.com/betaflight/betaflight, 2021.

[21] Esp32 xbee homepage. https://github.com/nebkat/esp32-xbee, 2021.

[22] Huawei. Huawei liteos homepage. https://gitee.com/liteos, 2021.

[23] C-K Kang, H. R. Mendis, and et al. Everything leaves footprints: Hardware accelerated intermittent deep inference,. In *TCAD*, 2020.

[24] Meriem Bettayeb, Qassim Nasir, and Manar Abu Talib. Firmware update attacks and security for iot devices: Survey. In *ArabWIC 2019*.

[25] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli. Secure firmware updates for constrained iot devices using open standards: A reality check. *IEEE Access*, 2019.

[26] A. Liu and P. Ning. Tinyecc: A configurable library for elliptic curve cryptography in wireless sensor networks. In *IPSN*, 2008.

[27] A. Shoufan and N. Huber. A fast hash tree generator for merkle signature scheme. In *ISCAS*, 2010.

[28] W. Dong, B. Mo, and et al. R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In *INFOCOM*, 2013.

[29] H. Tseng, L. M. Grupp, and S. Swanson. Underpowering nand flash: Profits and perils. In *DAC*.

[30] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM TOCS*, 2014.