WoFS: A Write-only File System for Privacy-aware Wireless Sensor Networks

Philipp Jahn*, Markus Sommer*, Artur Sterz*, Jonas Höchst†, Bernd Freisleben*

*Dept. of Mathematics & Computer Science, University of Marburg, Germany
E-Mail: {jahnph, msommer, sterz, freisleb}@informatik.uni-marburg.de

†tRackIT Systems GmbH, Cölbe, Germany
E-Mail: hoechst@trackit.systems

***Abstract*—Wireless sensor networks (WSNs) can automate data sensing tasks. To ensure redundancy and manage network connectivity issues, a sensing node stores a copy of the gathered data. Since this data may contain sensitive personal or business information, protecting privacy and preventing unauthorized access is crucial. We introduce the Write-only File System (WoFS), a novel encryption system for WSNs that secures data without user interaction, even if a sensor node is stolen. WoFS utilizes either symmetric encryption with volatile keys via a ratchet mechanism or asymmetric encryption. Asymmetric encryption, while slower, allows operation post-reboot, unlike the ratchet-based method. Our experiments show that WoFS achieves write speeds of 200 MB/s or higher, making it suitable for WSN applications. All developed software and artifacts are available under a permissive open-source license.**

***Index Terms*—Data Protection**

I. Introduction

Recent miniaturization and proliferation of affordable microchip systems have revolutionized environmental data collection. Wireless sensor networks (WSN) enable autonomous sensing, reducing manual data collection. However, in some scenarios, wireless connections may lack throughput, are costly, or are unavailable due to poor connectivity [1]. Consequently, data is often stored on the sensing node itself. While most sensor data does not contain human-related information, some sensors might inadvertently collect privacy-sensitive data. This can occur during wildlife recordings or intentionally in urban analyses like traffic flow studies [1]. Such situations may make it difficult to gain public approval for data collection in sensitive areas, such as city parks and recreational spaces.

Some environmental monitoring devices, like BatRack [2] for autonomous bat call detection, record audio and video in public forests, potentially violating visitors' personal rights. Additionally, the collected data belongs to WSN operators. Even sensor nodes that do not record personal data may contain protected information relevant for planning, such as in wind power expansion [3]. Competing groups, including researchers or activists, could steal this data for their own use. Therefore, measures should be taken to: (i) protect individuals' personal rights, (ii) prevent unlawful data use by third parties, and (iii) address concerns of citizens exposed to WSN data sensing.

In this paper, we present a novel encryption system for WSNs, called Write-only File System (WoFS), which protects data without requiring user interaction and even if a sensor node is stolen. WoFS is a hybrid encryption system using volatile keys either utilizing asymmetric encryption or a ratchet mechanism with state-of-the-art cryptographic algorithms. During provisioning, public and initialization keys are stored on sensor nodes, while private keys remain on the provisioning system. This approach ensures that private keys are never stored on sensor nodes, making the WSN autonomous and secure. The results of our evaluation show that WoFS is generally capable of achieving write speeds of 200 MB/s or more, which is plenty for WSN applications. We release the source code [1] together with all scripts [2] and experimental artifacts [3] under permissive open-source licenses.

II. Related Work

Privacy and data protection are crucial in the digital age, typically addressed through effective encryption methods. Major smartphone platforms like Google Android and Apple iOS use sophisticated hardware/software combinations to encrypt different data areas with separate keys by default [4]. Operating systems employ BitLocker (Windows) [5], FileVault (macOS) [6], and LUKS (Linux) [7] for system-wide user data encryption. While these offer suitable privacy levels, they require user interaction (e.g., passphrase entry), making them unsuitable for our purpose. Several approaches have been proposed for disk and file encryption. Encrypted overlay filesystems map a virtual cleartext folder to an encrypted one. *goCryptFS* uses symmetric encryption for data and metadata via the FUSE API [8]. *eCryptfs*, a kernel module implementation, offers better performance and supports easier file sharing [9]. *TransCrypt* facilitates multi-user usage with separate keys per file, stored using an asymmetric key system for user-grained access [10]. *DEFY* offers encryption and plausible deniability of encrypted data's existence, leveraging NAND flash drive operations, but risks data loss at higher deniability levels [11]. Martin [12] proposed a system based on

[1] https://github.com/umr-ds/wofs
[2] https://github.com/umr-ds/wofs-eval
[3] https://zenodo.org/records/12819658

combining asymmetric and symmetric encryption motivated by its use in the military sector [12]. *Differential privacy* adds randomly generated noise to true measurements, enabling accurate database information while ensuring high privacy levels [13]. However, this approach still leaves devices vulnerable to information gathering if stolen. Offloading techniques for reducing encryption overheads reach security and low latency requirements of industrial Internet of Things (IoT) applications with well-established standard ciphers [14]. However, the main objective is to reduce overheads, not to provide privacy in case of device loss or theft. Various approaches have been developed to enhance privacy in specific domains like healthcare IoT [15], smart grid applications [16], and FPGA encryption [17]. These methods leverage domain-specific characteristics to achieve high confidentiality levels. However, such solutions often lack generalizability.

III. Design

A. Threat Model

We assume that WSN nodes operate in remote locations, requiring autonomous operation without user input. Thus, WoFS must set up all cryptographic primitives independently. This, coupled with the risk of private key compromise, necessitates that no private cryptographic keys are deployed on nodes. We consider two attack vectors: remote connection and physical access. For remote attacks with limited privileges, we must prevent access to data being written to storage. Root-level remote access, allowing memory dumps containing encryption keys, cannot be handled. Physical access while powered on could potentially access system memory, but our threat model focuses on petty theft where devices are removed and powered off. It is crucial to prevent data extraction from stolen nodes or storage while ensuring system operability after power loss-induced reboots. Purely symmetric encryption methods are unsuitable, since keys would be inaccessible or deleted.

B. WoFS Properties

The writing process can write data to disk, but cannot read it back. The original data can only be restored with the master key k^m. Therefore, WoFS is designed as an overlay append-only file system. This way, it can easily be used with existing programs, which only have to write to the WoFS file system.

WoFS has a write-mode and a read-mode. The write-mode works without the master key k^m and is the mode during data collection and storage. The read-mode requires the master key k^m and is used to retrieve data. The master key k^m is not stored on the node, and the derived decryption key k must only be recoverable if the master key k^m is available. This ensures that no information for decryption can be gathered after an attacker steals a node. Metadata such as the file name, owner, and file size could hint at the file content, possibly leaking sensitive information. Therefore, metadata is obfuscated. Due to its append-only nature and metadata obfuscation, WoFS only provides a subset of file system functionality.

In write-mode, the content of the root directory cannot be read, making checking for a file's existence impossible. Thus, duplicate plain text file names need to be handled. This is solved by appending the number of the inode to the filename. Furthermore, extended attributes are not supported, since this metadata is stripped during encryption. Symbolic and hard links are also not supported, since the existence of their target cannot be verified.

C. Cryptographic Methods

As discussed in Section III-A, a WSN node should be operational after a reboot, but must protect data when stolen. We employ asymmetric encryption, WoFS-AE, which, due to multiple encryption processes, is slower in write speeds than symmetric encryption. Therefore, we also present WoFS-RE, using ratchet-based symmetric encryption. However, WoFS-RE cannot continue operation after a reboot, since all keys are in volatile memory. This flexibility allows WoFS to support the most appropriate encryption approach based on the specific threat model and requirements.

1) WoFS-AE (using asymmetric encryption): Using WoFS-AE, a public/private keypair k^m_{pub} and k^m_{priv} is required. The private key k^m_{priv} must never be stored on a node of a WSN. In write-mode, WoFS-AE only needs access to the public key k^m_{pub}. Subsequently, for each write of data d_i, a new symmetric key k_i is created to encrypt d_i. Symmetric key k_i is then encrypted using the public key k^m_{pub} and stored with the data d_i. This means that an unprivileged process cannot access any data written by another process, protecting against basic remote-access-attacks. Read-mode requires the private key k^m_{priv}. Keys $k_{1,\dots i,\dots n}$ are decrypted and used to decrypt the data $d_{1,\dots,i,\dots n}$.

2) WoFS-RE (using a ratchet mechanism): WoFS-RE has two keys: the master key k^m and a chain key k^c_1. The master key k^m is is used to derive all other keys and must not be stored on the node. When a new encryption key k_i is needed, the chain key k^c is used to derive two new keys: a new chain key k^c_{i+1} and the encryption key k_i. The encryption key k_i can now be used to encrypt d_i and must be discarded immediately afterwards. The chain keys can only be generated in the direction from the master key k^m to the most recent chain key k^c_j, but not the other way around. Thus, if an attacker observes one chain key k^c_j, all future keys are compromised, but previous keys remain safe.

3) Authentication: WoFS is only designed to protect data at rest. It cannot be used to ensure that a trusted source has written the data. Authenticating data is out-of-scope of this work. WoFS does support *authenticated encryption* which prevents cyphertext from being changed after the fact.

IV. Implementation

WoFS uses FUSE and is written in the C language.

A. FUSE

FUSE is a file system kernel module for Linux that allows writing file systems as user space programs instead of kernel modules. WoFS is implemented using FUSE for simplicity and to enable the use of high-level cryptographic APIs.

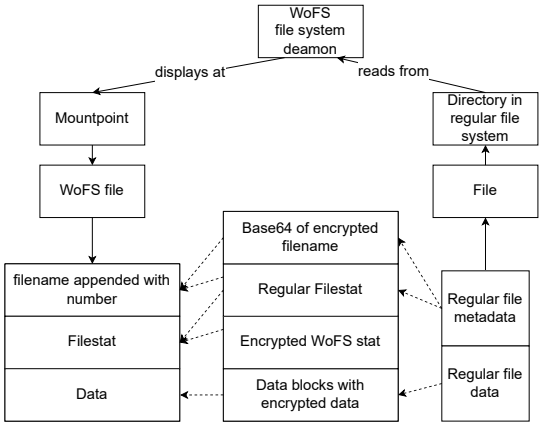

Fig. 1. Architecture of WoFS

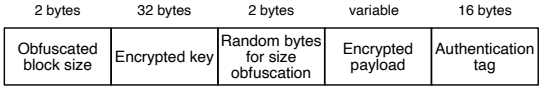

Fig. 2. Blocks

B. File System

The `write()` and `read()` syscalls are not directly passed from the kernel to WoFS, but are forwarded from the FUSE kernel module. Furthermore, WoFS does not interface directly with a block device, but stores data in files on the file system it is mounted on, as shown in Figure 1. For each file that is created in WoFS, one file is created in the underlying file system. The encrypted file contains encrypted metadata, the state needed for decryption, and the encrypted data.

C. Writing Blocks

Data is written in variable-size blocks. The buffer from each `write()` call is split into blocks for encryption. Each block starts with two bytes of block size, followed by the encrypted data and encryption overhead, as shown in Figure 2. The layout differs for WoFS-AE and WoFS-RE. For write calls smaller than the maximum payload size, one block of matching size is generated. For larger write calls, the payload is split into several blocks. The size field ensures the correct amount of data is supplied to the decryption function. Due to authentication, arbitrary sub-streams of the file cannot be decrypted directly.

D. Reading Blocks

Files can only be read sequentially from beginning to end. The requested data boundaries from the `read()` syscall may not align with WoFS block boundaries. Thus, all blocks containing the requested data must be decrypted, even if only a portion of a block is needed.

E. Metadata Obfuscation

WoFS encrypts the file size, permissions, and user and group ID. Other metadata, such as access times, are taken from the encrypted file. The original metadata is at the beginning of the encrypted file, and the file size is updated when data is written. File ownership and permissions are set at creation and cannot be updated later, as WoFS is append-only. To decrypt files, each block's size is written into the encrypted file. However, to prevent attackers from calculating the original file size, WoFS obfuscates the block size by encrypting two random bytes in each block and XOR'ing the block size with these bytes from the last block.

F. Cryptographic Primitives and APIs

To reduce attack vectors, we use the well-known cryptography library *libsodium*[4]. Libsodium employs *Curve25519* elliptic curve cryptography for public key encryption and *XSalsa20-Poly1305* for authenticated encryption. The elliptic curve method reduces overhead compared to non-elliptic algorithms like RSA. The *XSalsa20* stream cipher avoids padding, while *Poly1305* provides authentication.

WoFS-RE uses *HKDF-SHA256* as the key derivation function for the ratchet mechanism. *HKDF-SHA256* is a HMAC-based function that derives keys using a key and a context. For each write call, the *HKDF-SHA256* derived key is used. Instead of an encrypted key, data is appended to a randomly generated nonce. The data is encrypted using *XChaCha20-Poly1305* for authenticated encryption.

V. Evaluation

Encryption introduces overhead regarding runtime, data rate, occupied space on disk, CPU, and memory. Therefore, we evaluated WoFS to quantify the additional effort and assess WoFS's practicability.

A. Experimental Setup

1) Data Source and Destination: To evaluate WoFS, we use the `dd` command[5]. `dd` allows writing of data to and from arbitrary locations. To avoid the limitations of slow disk reads and writes, we get data from `/dev/urandom`[6], which constantly returns pseudo-random numbers. Furthermore, we use `nullfs`[7] as the destination. `nullfs` behaves in the same way as writing data to `/dev/null`, i.e., it will not write any data anywhere but simply return from the write call, making the CPU the limiting factor and not an HDD or SSD.

2) Experimental Parameters:

a) Encryption Software: The first parameter of our experiments is the used encryption software. Besides WoFS-AE and WoFS-RE, we also evaluated two alternatives: eCryptfs[8], a Linux kernel module, and EncFS[9], which is also a FUSE-based encrypted file system. For eCryptfs, the experimental data was written to disk instead of `nullfs`, because Linux does not allow to mount two file systems to the same location. The possible alternative, using a RAM disk, was also not

[4]https://doc.libsodium.org
[5]https://man7.org/linux/man-pages/man1/dd.1.html
[6]https://man7.org/linux/man-pages/man4/random.4.html
[7]https://github.com/abbbi/nullfsvfs
[8]https://www.ecryptfs.org
[9]https://github.com/vgough/encfs

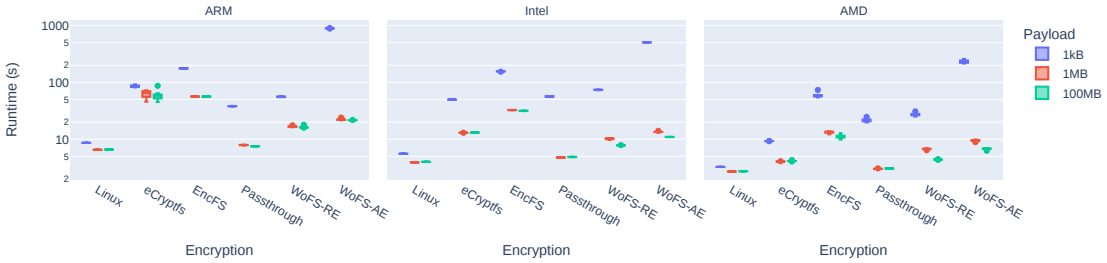

Fig. 3. Time to perform the experiments (on a logarithmic y-axis)

TABLE I
FILE SPACE OCCUPIED ON DISK

Version	1 kB	1 MB	100 MB
Linux		1,000,000,000 B (1 GB)	
eCryptfs		1,000,009,728 B	
EncFS		1,000,000,008 B	
WoFS-RE	1,019,000,084 B	1,000,304,084 B	1,000,308,074 B
WoFS-AE	1,050,000,068 B	1,000,800,068 B	1,000,810,568 B

feasible due to the harsh memory restrictions of the device used for the ARM-based experiments (see the targets below). Furthermore, we implemented a minimal file system that does not encrypt anything but passes data through FUSE. This allows us to estimate the overhead introduced by FUSE itself without any encryption. Finally, we also performed direct-write experiments without FUSE, which serve as a benchmark.

b) Payload Size: To assess the impact of context switches between kernel and user space, we use three different payload sizes: 1 kB, 1 MB, and 100 MB. The larger the payload size, the more data will be written per chunk, reducing context switches. In each experiment, a total of 1 GB of data is copied from `/dev/urandom` to `nullfs`. With 1 kB blocks, 1 million blocks are written; with 1 MB blocks, 1000 blocks are written; and with 100 MB blocks, 10 blocks are written.

c) Targets: To quantify the impact of different CPUs, three different targets were used: ARM Cortex A72 1.5 GHz `aarch64` denoted as ARM, a `x86_64` Intel Core-2-Duo 2.4 GHz CPU (denoted as Intel) and a relatively modern AMD EPYC 7742 64-Core 2.25 GHz `x86_64` CPU, denoted as AMD. This selection of CPUs is used as stand-ins for low-power (ARM), mid-range (Intel), and high-end (AMD) CPUs.

d) Repetitions: We performed each experiment 10 times, resulting in 540 experiments in total.

3) Metrics:

a) Runtime: `dd` itself reports the time needed in seconds, which we use to evaluate the time it takes to copy 1 GB of data from `/dev/urandom` to `nullfs`.

b) Data Rate: We measured the data rate by the number of bytes written per second. `dd` reports the number of MB/s as the result of the total number of bytes written divided by the time it took to copy the data.

c) Space on Disk: The space requirements on disk were gathered using `ls -l` after the experiment was done and the folders were unmounted from their encryption software.

d) CPU: CPU statistics were collected from the `/proc/pid/stat` file, where the number of scheduling cycles is reported on request.

e) Memory: Finally, we collected memory-related statistics from the `/proc/pid/smaps` file, where the kernel reports how many pages in the virtual memory are occupied by the given process.

B. Runtime

When writing data, it is important that the operation is executed reasonably fast, avoiding blocking process execution due to I/O waiting. Figure 3 shows the runtime results for the experiments. On the x-axes, the used software is shown, while the y-axes show the time it took to complete the `dd` copy process on a logarithmic scale. Each subplot shows the results for the different targets, and the color indicates the payload size. The lower the box, the better.

There are five main observations. First, the modern AMD CPU is the fastest, Intel is second, and the ARM CPU is the slowest. The most noticeable difference is in the 1 kB WoFS-AE experiments: the AMD CPU completes in a median of about 240 seconds, Intel in about 500 seconds, and ARM in about 903 seconds. This pattern is consistent across all software and payload sizes, even in plain Linux experiments.

Second, the Linux-only experiments are the fastest across all targets and payload sizes. This is expected, since the `dd` process writes data to `nullfs` without FUSE or encryption overhead. Comparing Linux to passthrough experiments, FUSE adds about 15% to 20% overhead for 1 MB and 100 MB experiments, while WoFS adds an additional 20% to 25%, depending on the CPU.

Third, the 1 kB experiments are the slowest. While this is also visible in the Linux-only experiments, it is especially apparent in the other experiments. This, together with the observation that FUSE only introduces an overhead of 15% to 20% for 1 MB and 100 MB experiments, shows that writing many small files should be avoided. This is also expected, since writing small blocks requires many context changes between kernel and user space. Additionally, WoFS-AE generates a key for every `write()` syscall and encrypts the data separately; many calls for `write()` with small buffers introduce additional context switches.

Fourth, experiments using WoFS-AE require the longest time. This is also expected, since every `write()` call causes

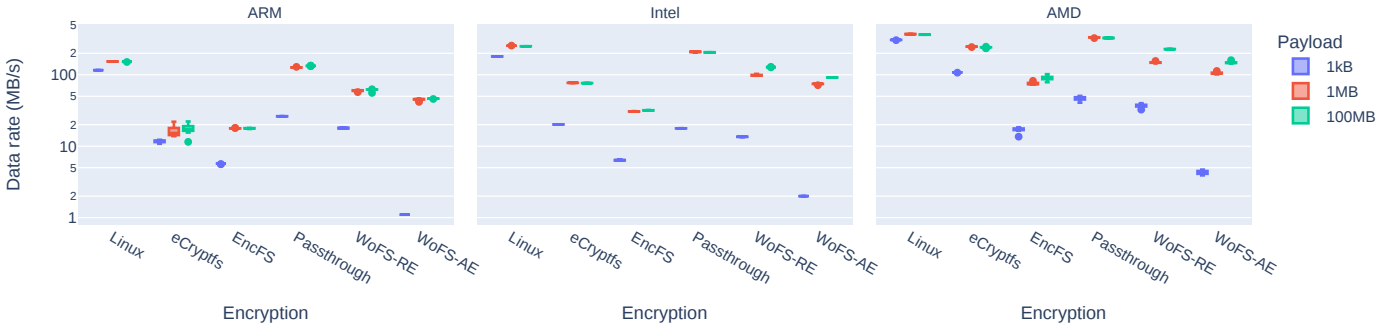

Fig. 4. Data rate of write operations

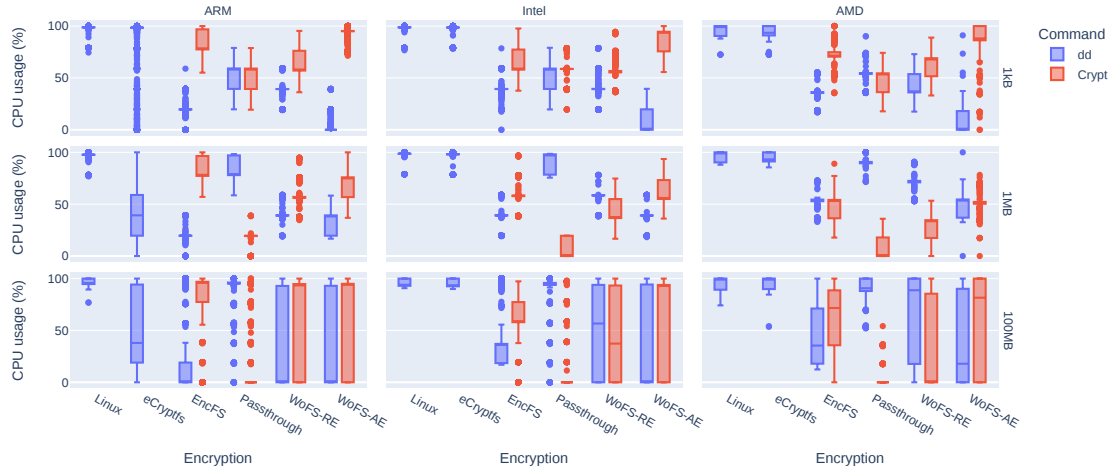

Fig. 5. CPU usage for different experiments

two encryptions: (i) the data is encrypted using the symmetric key, and (ii) the symmetric key is encrypted using the public asymmetric key.

Among FUSE-based encryption software, EncFS is the slowest for 1 MB and 100 MB experiments. eCryptfs outperforms all FUSE-based software except passthrough, despite being the only one writing to disk instead of `nullfs`.

C. Data Rate

The pure runtime shows how long it takes to write a certain amount of data, but not what it means in terms of write speeds of disks. Therefore, we also evaluated the data rate as the number of bytes written per second and relate this to the write speeds of different disk types. Figure 4 shows the achieved data rate in MB/s for each experiment. The y-axes show the achieved data rate in MB/s, the rest of the figure is the same as in Figure 3 First, all observations made in the runtime evaluation can be translated to the data rate. For example, the AMD CPU achieves the highest write speeds, up to 370 MB/s, Intel up to 256 MB/s, and ARM about 151 MB/s as medians. Linux-only experiments are the fastest, with 1 MB and 100 MB passthrough experiments about 15% to 20% slower. The 1 kB experiments show the lowest data rates due to frequent context switches, with WoFS-AE being particularly slow. A key takeaway is to use the largest possible buffer for the `write()` syscall to minimize context switches and maintain reasonable write speeds.

WoFS-RE on the ARM CPU achieves a median of 58 MB/s to 62 MB/s, while WoFS-AE is around 45 MB/s, comparable to USB-2 speeds and adequate for most WSN applications. Even in high-demand scenarios, WoFS maintains sufficient write speeds. We anticipate that WSNs in demanding applications will use more capable CPUs, improving write speeds. For instance, the AMD CPU exceeds 100 MB/s for 1 MB payloads with symmetric encryption and can reach up to 228 MB/s for 100 MB payloads using WoFS-RE.

D. Space on Disk

Each block written by WoFS-AE requires 52 bytes of additional data and 21 bytes for WoFS-RE due to the metadata required. Besides the overhead per block, WoFS-AE requires additional 118 bytes and WoFS-RE requires 79 bytes. Table I shows the required space on disk in bytes for the different experiments. With 5% more space on disk for 1 kB experiments, WoFS-AE has the highest overhead of all experiments. For 1 MB and 100 MB experiments, the overhead is only 0.8%. With about 1.9% overhead for 1 kB and 0.3% overhead for 1 MB and 100 MB experiments, WoFS-RE requires even less additional space on disk.

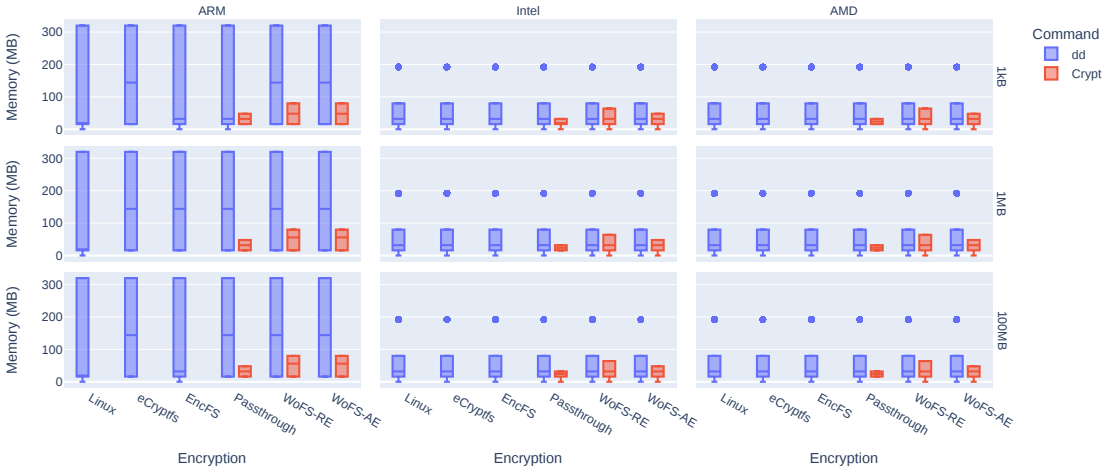

Fig. 6. Memory usage for different experiments

E. CPU

Since we use `/dev/urandom` as the data source and `/dev/null` (except for eCryptfs) as the data destination, we expect a CPU utilization of nearly 100%, since there is no waiting time for I/O. Therefore, we evaluated the CPU utilization to (i) test this hypothesis and (ii) to quantify where the CPU is used in the `dd` process or WoFS. Figure 5 shows the CPU utilization for each experiment. On the x-axes, the used software is shown, while the y-axes show the CPU utilization in %. Each column shows the results for the different targets, while the rows show the payload sizes. The color denotes either `dd` or WoFS. For the Linux-only experiments, the median CPU utilization is between 95% and 100% for the `dd` process, as expected. While the behavior of the different targets only varies in details, the different payload sizes significantly differ in their results. For 1 kB experiments with no encryption or WoFS-RE, `dd` requires about 40% CPU, WoFS the remaining 60%. However, WoFS-AE requires a median between 90% and 95%, indicating that I/O speed is not the bottleneck, but encryption. For 1 MB experiments with no encryption, `dd` requires most of the CPU and FUSE only takes a small portion of the CPU. For the Intel and the AMD CPUs, the median is even below 1%. The same applies to 100 MB experiments. One key finding in the 100 MB Intel and AMD experiments is that the share of the CPU utilization of the `dd` process is higher compared to WoFS-RE, but this changes for WoFS-AE, where WoFS requires the highest part of the CPU. For example, the median CPU utilization for `dd` using WoFS-RE with a payload size of 100 MB on the Intel CPU is at about 56% and 37% for WoFS, `dd` uses below 1% for the same experiment using WoFS-AE and WoFS about 93%. This supports the hypothesis that WoFS-AE achieves relatively poor results in terms of runtime and data rate compared to WoFS-RE due to additional encryption steps.

F. Memory

The last metric in our evaluation is memory utilization, as shown in Figure 6. The y-axes show the memory utilization in MB, the remainder of the figure is the same as Figure 5 Two main takeaways become apparent in terms of memory utilization. First, the memory requirements of the `dd` process are always the same, regardless of payload size and used software. Only the target makes a difference. The ARM CPU requires a median of about 168 MB, while the Intel and AMD CPUs require a median of about 56 MB of memory. The second takeaway is that WoFS always requires about the same amount of memory (i.e., between 32 MB and 56 MB). Passthrough experiments allocate less peak memory. For the Intel and AMD CPUs, this means a median overhead of about 87% to 100%. For the ARM CPU, the median overhead is at about 33%, because `dd` requires more memory than WoFS. However, nodes in modern WSNs are equipped with multiple GBs of memory. Even the Raspberry Pi 5 has at least 4 GB memory, making the additional 56 MB memory negligible.

VI. Conclusion

We presented WoFS, a novel privacy-aware file system for WSNs. It secures data without user interaction, even if a sensor node is stolen. WoFS uses symmetric encryption with volatile keys via a ratchet mechanism or asymmetric encryption. The latter results in slower write speeds, but it allows nodes to continue operating after a reboot, unlike the ratchet-based method. Our evaluation shows that WoFS achieves write speeds of 200 MB/s or more.

Future work includes exploring more efficient cryptographic primitives to reduce WoFS overhead, especially for small files, and developing a kernel module for increased efficiency. Additionally, WoFS should be adapted for microcontrollers and IoT devices, which are commonly used in WSNs, but lack an operating system with defined syscalls.

Acknowledgment

This work is funded by the Hessian State Ministry for Higher Education, Research and the Arts (HMWK) (LOEWE emergenCITY), and the German Research Foundation (DFG, Project 210487104 - SFB 1053 MAKI).

REFERENCES

[1] D. Zeuss, L. Bald, J. Gottwald, M. Becker, H. Bellafkir, J. Bendix, P. Bengel, L. T. Beumer, R. Brandl, M. Brändle, S. Dahlke, N. Farwig, B. Freisleben, N. Friess, L. Heidrich, S. Heuer, J. Höchst, H. Holzmann, P. Lampe, M. Leberecht, K. Lindner, J. F. Masello, J. Mielke-öglich, M. Mühling, T. Müller, A. Noskov, L. Opgenoorth, C. Peter, P. Quillfeldt, S. Rösner, R. Royaute, C. Mestre-Runge, D. Schabo, D. Schneider, B. Seeger, E. Shayle, R. Steinmetz, P. Tafo, M. Vogelbacher, S. Wöllauer, S. Younis, J. Zobel, and T. Nauss, "Nature 4.0: A networked sensor system for integrated biodiversity monitoring," *Global Change Biology*, vol. 30, no. 1, p. e17056, 2024.

[2] J. Gottwald, P. Lampe, J. Höchst, N. Friess, J. Maier, L. Leister, B. Neumann, T. Richter, B. Freisleben, and T. Nauss, "BatRack: An open-source multi-sensor device for wildlife research," *Methods in Ecology and Evolution*, Jul. 2021.

[3] J. Höchst, J. Gottwald, P. Lampe, J. Zobel, T. Nauss, R. Steinmetz, and B. Freisleben, "tRackIT OS: Open-source software for reliable VHF wildlife tracking," in *51. Jahrestagung der Gesellschaft für Informatik, Digitale Kulturen*, ser. LNI. GI, Sep. 2021.

[4] S. Garg and N. Baliyan, "Comparative analysis of Android and iOS from a security viewpoint," *Computer Science Review*, vol. 40, p. 100372, 2021.

[5] C. Tan, L. Zhang, and L. Bao, "A deep exploration of BitLocker encryption and security analysis," in *20th International Conference on Communication Technology*. IEEE, 2020, pp. 1070–1074.

[6] O. Choudary, F. Grobert, and J. Metz, "Security analysis and decryption of Filevault 2," in *9th IFIP International Conference on Digital Forensics*. Springer, 2013, pp. 349–363.

[7] S. Bossi and A. Visconti, "What users should know about full disk encryption based on LUKS," in *14th International Conference on Cryptology and Network Security*. Springer, 2015, pp. 225–237.

[8] T. Hornby, "Security Audit of gocryptfs v1.2," Mar. 2017. [Online]. Available: https://defuse.ca/audits/gocryptfs.htm

[9] M. A. Halcrow, "eCryptfs: An enterprise-class encrypted filesystem for Linux," in *Proceedings of the 2005 Linux Symposium*, vol. 1, 2005, pp. 201–218.

[10] S. Sharma, "Transcrypt: Design of a secure and transparent encrypting file system," *M. Tech Thesis, Indian Institute of Technology Kanpur, Kanpur*, 2006.

[11] T. M. Peters, M. A. Gondree, and Z. N. Peterson, "Defy: A deniable, encrypted file system for log-structured storage," 2015.

[12] T. Martin, "Undecryptable symmetric encryption," in *GCC Conference and Exhibition*. IEEE, 2011, pp. 225–228.

[13] C. Dwork, "Differential privacy," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2006, pp. 1–12.

[14] J. Hiller, M. Henze, M. Serror, E. Wagner, J. N. Richter, and K. Wehrle, "Secure low latency communication for constrained industrial IoT scenarios," in *43rd Conference on Local Computer Networks*. IEEE, 2018, pp. 614–622.

[15] M. Min, X. Wan, L. Xiao, Y. Chen, M. Xia, D. Wu, and H. Dai, "Learning-based privacy-aware offloading for healthcare IoT with energy harvesting," *Internet of Things Journal*, vol. 6, no. 3, pp. 4307–4316, 2018.

[16] G. Giaconi, D. Gunduz, and H. V. Poor, "Privacy-aware smart metering: Progress and challenges," *Signal Processing Magazine*, vol. 35, no. 6, pp. 59–78, 2018.

[17] G. Cano-Quiveu, P. Ruiz-de-clavijo Vazquez, M. J. Bellido, J. Juan-Chico, J. Viejo-Cortes, D. Guerrero-Martos, and E. Ostua-Aranguena, "Embedded LUKS (E-LUKS): a hardware solution to IoT security," *Electronics*, vol. 10, no. 23, p. 3036, 2021.