Codo: Confidential Data Storage for Wireless Sensor Networks

Ibrahim Ethem Bagci[1], Mohammad Reza Pourmirza[1], Shahid Raza[2], Utz Roedig[1], Thiemo Voigt[2,3]
[1]School of Computing and Communications, Lancaster University, Lancaster, UK
{i.bagci, m.pourmirza, u.roedig}@lancaster.ac.uk
[2]Swedish Institute of Computer Science, Kista, Sweden
{shahid, thiemo}@sics.se
[3]Uppsala University, Sweden

***Abstract*—Many Wireless Sensor Networks (WSNs) are used to collect and process confidential information. Confidentiality must be ensured at all times and, for example, solutions for confidential communication, processing or storage are required. To date, the research community has addressed mainly the issue of confidential communication. Efficient solutions for cryptographically secured communication and associated key exchange in WSNs exist. Many WSN applications, however, rely heavily on available on-node storage space and therefore it is essential to ensure the confidentiality of stored data as well. In this paper we present Codo, a confidential data storage solution which balances platform, performance and security requirements. We implement Codo for the Contiki WSN operating system and evaluate its performance.**

I. Introduction

In many Wireless Sensor Networks (WSNs) sensor data is transferred immediately to a sink for further processing. In such applications no data is stored on nodes and confidentiality of stored information is not an issue. A number of (recent) applications use, however, available on-node storage space to add new features or to improve network performance. For example, on-node storage may be used when the network capacity is insufficient to transport all gathered data from all nodes to the sink. Instead, stored data is pre-processed by nodes and only processing results are transmitted. Moreover, the sink may as well request processing or transmission of stored data.

An example of such an application is industrial process monitoring and control [1] where sensors used to collect and store vast amounts of data on production processes. Nodes process sensor data and transmit results to a sink node. At times, the sink may request to process sensor data over specific time periods (e.g. to calculate the average of a sensor reading over a recent set time period) or request all sampling points in a specific time period. Such specific data requests may be necessary for error diagnosis or to calibrate the overall production processes. As sensors store information about production processes it is of vital interest to a company to keep such information hidden from competitors. If a node is removed from the facility it should not be possible to retrieve the stored data.

To secure data stored on nodes it has been proposed to simply encrypt the data before storage using key chains (see, for example, [2], [3]). Such a naive approach ensures confidentiality but at the same time restricts a node's processing capability and does not consider system performance issues. Such existing solutions do not enable nodes to access already stored data for in-network data processing on nodes and do not allow us to balance performance and security concerns. Furthermore, existing solutions are not tailored to hardware specifics such as flash memory layout or available hardware support for cryptographic algorithms.

In this paper we present Codo, a framework for efficient confidential data storage on sensor nodes. Codo addresses the aforementioned shortcomings present in existing solutions by enabling confidential data storage and in-network data processing on nodes. Security concerns and performance can be balanced by deciding how much unencrypted data can be present on a node at any given point in time. To improve performance, encrypted data storage is aligned with flash memory layout and cryptographic hardware support. The specific contributions of the paper are:

- *Codo:* We give a design specification of the efficient confidential data storage framework.
- *Codo Implementation:* We detail the implementation of Codo for the Contiki OS [4] running on a Tmote Sky. In particular, we show the integration of Codo with the Contiki flash file system Coffee [5].
- *Codo Evaluation:* We evaluate the different aspects of Codo and quantify performance implications. Where possible we compare results with existing traditional approaches.

The next section discusses related work. Section III describes the proposed confidential storage framework Codo. We describe our Codo implementation for the Contiki operating system in Section IV. Section V presents the evaluation of Codo and Section VI concludes the paper.

II. Related Work

Bhatnagar and Miller [2], Pietro et al. [6] and Girao et al. [7] present a secure and reliable file systems. All of these solutions use irreversible key generation methods which prevent nodes

to access data locally. Ren et al. propose a secure, dependable and distributed storage scheme using public key encryption to ensure data confidentiality [3]. Their scheme does not allow nodes to access stored data either. In contrast, Codo allows nodes to access stored data for processing. In addition, public key cryptography requires more processing than symmetric cryptography as used in Codo.

Due to higher speed and better security (e.g. resistant to cold boot attacks) hardware based storage encryption became available lately by many vendors. For example, the company Ironkey [8] manufactures secure USB flash drives in which AES 256-bit encryption in CBC mode is implemented in hardware. Codo is different to these solutions as it does not rely on special storage hardware.

III. Confidential Data Storage

A. Limitations of Existing Solutions

Existing security solutions focus on optimizing encryption performance (for example, by optimizing encryption algorithms [9] or by employing specialised cryptographic hardware [10]). However, performance gains that result from looking at secure storage from a systems perspective are largely ignored. In existing solutions data is generally encrypted as soon as it produced (for example, in [2], [3], [6], [7]). However, for many applications it is from a security perspective possible to cache data unencrypted before performing bulk encryption and storage. Obviously, the amount of data that can be cached unencrypted will depend on the specific application.

Current solutions are hardware agnostic which leads to inefficiencies. Flash memories used on sensor nodes are restricted in terms of read and write capabilities. It is often only possible to access data in chunks rather than in individual bytes and it is always more efficient to process data in chunks. Thus, crypto mechanisms should operate on chunk sizes that reflect hardware capabilities. Existing solutions also ignore that hardware support for cryptographic operations exists on WSN nodes. Hardware support is generally available for secure communication but it is possible to re-use these features for secure storage. Again, this cryptographic hardware is optimised for specific data chunk sizes and if used in the context of storage these hardware restrictions must be taken into account.

B. Codo: Confidential Data Storage Framework

Codo tackles the aforementioned limitations and shortcomings of existing confidential data storage solutions. The framework aims to realize confidential data storage with minimal impact on node operation and performance.

In our storage framework data is organised in *DataChunks*. Some unencrypted data is cached to improve performance; depending on application security and performance needs it can be decided how much unencrypted cached data can be present. To improve performance only complete DataChunks are cryptographically processed. The size of DataChunks is matched to the capabilities of storage hardware (e.g. page sizes) and to the capabilities of the encryption hardware (e.g. buffer size of cryptographic processor).

DataChunk Size: The DataChunk size S_D is determined by a number of factors. These are:

- Cryptographic Algorithm: The cryptographic algorithm usually operates on fixed block sizes S_B. The DataChunk size should therefore be aligned with this block size. Thus, the DataChunk size S_D must be a multiple of S_B.
- Cryptographic Hardware Support: If cryptographic hardware support is available it is normally operating most efficiently on a block size of S_C. The transfer of data to and from the crypto processor has a fixed cost element (addressing, loading operations, etc.) and a variable cost element that depends on the data size to be processed. S_C is a multiple of S_B and in many practical setting $S_C = S_B$. Again, S_D must be a multiple of S_C.
- Flash Memory: Flash memory is organized in pages of size S_P. Depending on the flash memory hardware the page size implies different constraints. For example, with some hardware it is only possible to read or write a whole page. Other hardware allows to read or write parts of a page but writing or reading of complete pages is most efficient (as the fixed cost of addressing has to be paid only once for all data associated with the page). It is therefore reasonable to align the DataChunk size with the page size. S_P should be therefore a multiple of S_D.

$$S_P = aS_D = bS_B = bS_C$$

$$b \equiv 0 \ (mod\, a)$$

$$\forall\, a, b\, \epsilon\, 1N^+$$

Data Caching: To increase the performance of the system, unencrypted DataChunks are cached. Write operations are cached and encryption is carried out after a certain amount of DataChunks are accumulated. Likewise, if previously stored data must be read/modified the complete corresponding DataChunk is decrypted and cached. Increasing the size of the cache leads to better performance, however with this approach the amount of unencrypted data present in the system would be bigger. Decreasing the size of the cache leads to better security, but at this time the system performance decreases. The number of unencrypted DataChunks N_D allowed in the system at any given time is a configuration parameter.

Key Management: In the current implementation of our framework, encryption keys are pre-shared. Future implementations might be enhanced with dynamic key management protocols such as IKE [11].

IV. Codo Implementation

Codo is implemented as an extension of Contiki's [4] Coffee Filesystem (CFS) [5]. CFS organizes files as a collection of similar sized pages (see Figure 1) that have generally the same size as the underlying flash memory pages[1]. For each

[1]It is possible to map several Contiki file system pages into one flash memory page. However, this only makes sense if the flash memory hardware is able to support operations on parts of a page.

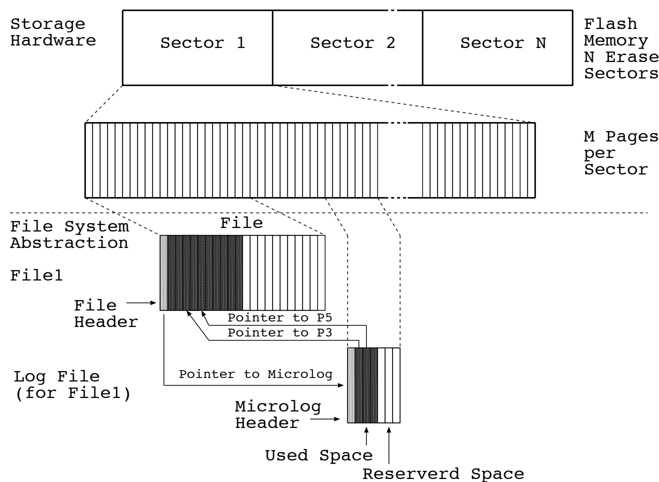

Figure 1. The Contiki Coffee File System (CFS)

new file a header is created and a number of consecutive free pages are allocated. New data is directly written to the empty pages in the file. If pages with existing content are modified a so called micro log file (also simply referred to as log file) is used. For modifications a micro log file is created and linked with the original file which contains a sequence of log records that have the same size as a page in the original file. Each log record points to the original page in the file and contains the updated information. If data is accessed, the CFS checks first if newer data is available in the log file before accessing the original file. After a certain amount of changes the log file is filled and *merged* with the original file to form a new consolidated file. The old file and micro log file are marked for garbage collection. The micro log structure is used because the flash memory hardware does not allow us to overwrite pages directly. Before overwriting a hardware page it is necessary to format and clear an entire erase sector containing many pages. This would be inefficient when used frequently and the use of a log file reduces erase sector formats to a minimum performed at convenient times by the CFS garbage collection. The CFS exposes standard functions such as `cfs_open()`, `cfs_write()`, `cfs_read()`, `cfs_seek()` and `cfs_close()` to the application for interaction with the filesystem.

A. Codo Extensions for CFS

To implement Codo with Contiki's CFS it is necessary to *(i)* modify and extend function calls provided by the existing CFS library and to *(ii)* modify and extend the behavior of internal CFS components. We detail these necessary modifications in the next paragraphs.

1) CFS Function Calls : Algorithm 1 shows a simple Contiki program that uses the CFS library. The definition in line 7, 13 and 15 is used to switch the API semantic between CFS with Codo (CFS_CRYPT) and standard CFS.

Without CFS_CRYPT a file is opened in line 6 using `cfs_open()` for reading which is indicated via the flag `CFS_READ`. In line 14 `cfs_read()` is used to read data from the file. `cfs_close()` (line 16) is used to close the open file.

With CFS_CRYPT, if data to be read using `cfs_read()` (line 9) is not yet available from the micro log file, the security manager component (see next section for details) is queried to provide the key required to decrypt the next DataChunk. The program leaves `cfs_read()` before completion of the read process and blocks on `ev == KEY_READY`. When the key is provided by the security manager a signal is sent to the waiting application process. When `cfs_read()` is now called again it will be able to decrypt the next data for which the key is now present. Multiple executions of `cfs_read()` with following `PROCESS_WAIT_UNTIL` may be necessary to complete one read as a sequence of different keys may be used. The call to `cfs_read()` with following `PROCESS_WAIT_UNTIL` (line 8 to line 12) can be combined within one *C* macro to hide the complexity of multiple function entries from the programmer.

`cfs_write()` is used in a similar way to `cfs_read()`.

`cfs_open()` supports the additional flag `CFS_NO_CRYPT` to indicate that a specific newly opened file should not be encrypted. Thus, the filesystem can hold encrypted and unencrypted files at the same time.

We also add two additional functions to the CFS API: `cfs_read_crypt()` and `cfs_write_crypt()`. These two functions can be used to read and write the encrypted data directly. If data is still in unencrypted form in the cache `cfs_read_crypt()` will perform encryption of the data. The security manager may have to be informed to provide necessary keys and multiple calls to `cfs_read_crypt()` followed by `PROCESS_WAIT_UNTIL` might be necessary. These two functions are particularly useful for situations in which encrypted data must be handled by the node. For example, with these functions it is possible to avoid re-encryption of data for data transport and the already securely stored data can be directly placed in network packets.

We provide a `cfs_merge()` which can be used to execute the processing costly merge of file and log file at a convenient time (for example, at times the system is idle).

`cfs_close()` is modified to ensure that a merge is executed which makes certain that all unencrypted cached data is encrypted and stored in the file. `cfs_close()` may require multiple function calls with `PROCESS_WAIT_UNTIL` as keys may have to be organized by the security manager for encryption.

2) CFS Components:

Micro Log: As described, the CFS uses the micro log files to handle flash memory read/write specifics. For the Codo implementation we modify the log file such that it becomes in addition a cache holding unencrypted DataChunks. In the standard CFS the log file is used for modifying write operations. In our CFS extension also read and initial write operations are operating on the log file.

Whenever data is read from the file it is first checked if the data is present in unencrypted form in the log file. If not, the key associated with the data is requested via the security

Algorithm 1 A simple Contiki application program using CFS with and without Codo extension.

```
[01] PROCESS_THREAD(cfs_test_process, ev, data) {
[02]  PROCESS_BEGIN();
[03]  char buf[100];
[04]  char *filename = "msg_file";
[05]  int fd; int n=0;
[06]  fd = cfs_open(filename, CFS_READ);
[07] #ifdef CFS_CRYPT
[08]  while(n<sizeof(buf)) {
[09]   n+=cfs_read(fd,buf+n, sizeof(buf));
[10]   if(n<sizeof(buf))
[11]       PROCESS_WAIT_UNTIL(ev == KEY_READY);
[12]  }
[13] #else
[14]  cfs_read(fd,buf, sizeof(buf));
[15] #endif
[16]  cfs_close(fd);
[17]  PROCESS_END();
[18] }
```

manager component and upon obtaining the key the data is decrypted and transferred to the log file. New data is always written to the log file. When the maximum size of the log file is reached the log file must be cleared and merged with the original file.

Security Manager: The security manager is implemented as a Contiki thread that is responsible to *(i)* generate new keys if needed *(ii)* communicate with the sink to store and retrieve keys.

The CFS can ask the security manager via a function call for a key to a specific DataChunk of a file. This request contains three parameters: file_id, DataChunk_id and flags. file_id is the filename which is a unique identifier, DataChunk_id is the number of the DataChunk for which a key is required. flags indicates if the requested key is for a portion of the file that has never been used before. If this is the case the security manager has two options. First, it can create the requested key, inform the filesystem and then transmit the key to the sink for storage. Second, it can send a request to the sink for a new key and when a response arrives inform the filesystem. If flags indicate that a key for a previously used DataChunk is needed and the key is not locally present, the security manager must send a request for the key to the sink.

Cryptographic Functions: For encryption/decryption we use AES in counter mode (CTR) with $128bit$ key length provided either by hardware (e.g. via the CC2420 radio chip present on many sensor node platforms) or by the open source MIRACL [12] library if hardware support is not available.

V. Codo Evaluation

We evaluate the Codo implementation based on Contiki's CFS using a Tmote Sky sensor node. To evaluate system performance we analyse the execution times of the CFS function calls. Execution times are important indicators as they are a measure for system responsiveness and are directly proportional to a node's energy consumption. Furthermore, we investigate the performance of data caching when the cache is located in flash memory or in RAM.

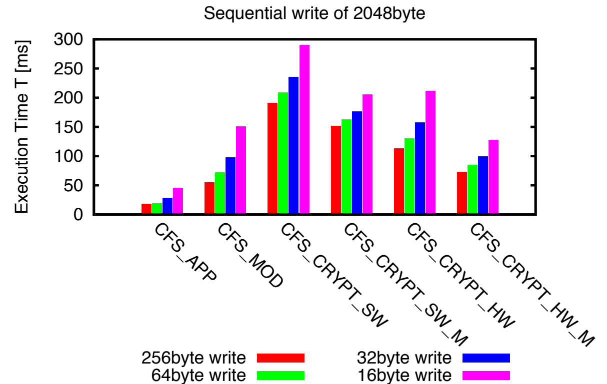

Figure 2. Writing of $2048byte$ in blocks of $256byte$, $64byte$, $32byte$ and $16byte$ using `cfs_write()`.

The cryptographic hardware support of the Tmote's CC2420 radio chip requires a minimum block size of $S_C = 16byte$. The Tmote provides an ST M25P80 flash memory with a page size of $S_P = 256byte$. We therefore select a DataChunk size of $S_D = 256byte$ to obtain a well matched system (see Section III). We use software (CFS_CRYPT_SW) or hardware (CFS_CRYPT_HW) supported encryption. The filesystem uses a log record size of $S_L = 256byte$ to match flash memory page size. Furthermore, the system is configured to use $N_L = 4$ log records which means that $S_L \cdot N_L = 1024byte$ of unencrypted data can be present on the system at any given point in time.

A. cfs_write() Performance

In this first experiment a file of size $2048byte$ is written using a sequence of `cfs_write()` calls. With each `cfs_write()` call S_W bytes are written to the file system ($S_W \in \{16, 32, 64, 256\}$). The execution time of each `cfs_write()` call is measured. The experiments are repeated using the original CFS in append mode (CFS_APP), the original CFS in modify mode (CFS_MOD) (data is appended, but the file is assumed as modified and therefore also the log file is used), Codo CFS with software (CFS_CRYPT_SW) and hardware supported (CFS_CRYPT_HW) encryption and Codo CFS with additional RAM supported log file (CFS_CRYPT_SW_M and CFS_CRYPT_HW_M). In this experiment the Security Manager holds the required key for the $2048byte$ sized file locally and there is no key exchange with the sink required. If keys are exchanged over the network key exchange times have to be added to the experiment results.

The experimental results are shown in Figure 2. Using CFS_APP and $S_W = 256byte$ the time to write all $2048byte$ to the file system is $18.3ms$. In this mode the file system does not make use of the log file structure and data is directly written to the file structure in flash memory. With CFS_MOD the time increases significantly to $54.8ms$ as the log file structure is involved in the writing process. Each write is directed to a log record in the log file in flash memory and when all log records are filled a merge is executed to integrate log file and original file. CFS_CRYPT_SW and CFS_CRYPT_HW are

write	CFS_APP	CFS_CRYPT_SW	CFS_CRYPT_HW	CFS_CRYPT_HW_M
1	2.29ms	4.06ms	4.12ms	1.43ms
2	2.29ms	3.14ms	3.17ms	1.40ms
3	2.29ms	2.93ms	3.02ms	1.40ms
4	2.29ms	2.93ms	2.93ms	1.37ms
5	2.29ms	168.67ms	90.39ms	63.45ms
6	2.29ms	3.14ms	3.14ms	1.40ms
7	2.29ms	2.99ms	2.96ms	1.37ms
8	2.29ms	2.93ms	2.96ms	1.37ms

Table I
WRITING OF $2048 byte$ IN 8 BLOCKS OF $256 byte$.

functionally identical to CFS_MOD but when the log file is merged with the original file, encryption has to be performed. Thus, with CFS_CRYPT_SW and CFS_CRYPT_HW execution times are $190.8ms$ and $112.7ms$. With RAM caching, the execution time reduces further to $151.6ms$ and $73.2ms$ (CFS_CRYPT_SW_M and CFS_CRYPT_HW_M).

The overall time of writing $2048byte$ to the file is not distributed equally among the 8 separate executions of `cfs_write()` with $S_W = 256byte$ (see Table I). The first `cfs_write()` takes for CFS_CRYPT_SW and CFS_CRYPT_HW slightly more time than the following three as some time to create the log file structure in flash memory is needed. The 5th write takes considerable more time than previous writes as the log file of size $N_L = 4$ is full and a merge must be executed before a log record can be written. During merge encryption is performed which requires significant processing time. The use of encryption hardware support improves encryption performance by 47%. With CFS_CRYPT_HW_M the first 4 write operations are faster than CFS_APP as data is written to the cache located in RAM.

When decreasing the write size S_W to $64bytes$, $32bytes$ and finally $16bytes$ the overall time necessary to write the file of $2048bytes$ increases (see Figure 2). This is expected as each `cfs_write()` call is associated with additional overhead. For example, the overall time to write a file of $2048bytes$ length increases from $112.7ms$ to $211.6ms$ when switching from $S_W = 256byte$ to $S_W = 16byte$ with CFS_CRYPT_HW. Note that CFS_CRYPT_HW_M outperforms CFS_MOD for $S_W = 16byte$. This means that under this condition Codo, which performs caching and encryption, outperforms the standard CFS when operating on files that have been modified.

Summary: The Codo CFS is relatively expensive in comparison to CFS. CFS does, however, not provide data confidentiality and this feature cannot be implemented at zero cost. For example, the overall execution time for writing $S_W = 256byte$ increases with Codo CFS (CFS_CRYPT_HW_M) compared to CFS (CFS_APP) by a factor of 4. However, individual write operations that do not require merging and encryption are faster with Codo (CFS_CRYPT_HW_M) (by a factor of 1.6 for the first write with $S_W = 256byte$). Furthermore, within an application scenario it might be possible to schedule costly merge and encrypt operations at times the system is idle and thus the overhead for providing confidentiality may not impact the overall system performance.

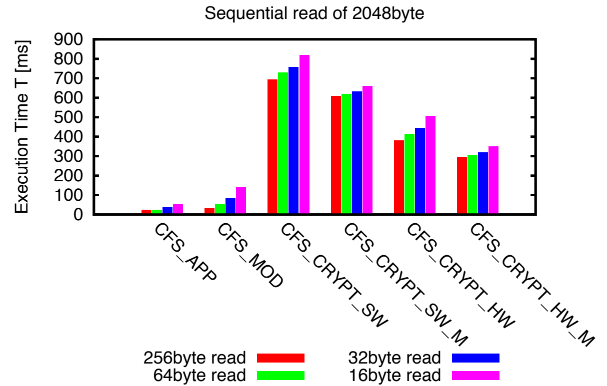

Figure 3. Reading of $2048byte$ in blocks of $256byte$, $64byte$, $32byte$ and $16byte$ using `cfs_read()`.

write	CFS_APP	CFS_CRYPT_SW	CFS_CRYPT_HW	CFS_CRYPT_HW_M
1	3.14ms	232.03ms	134.31ms	100.56ms
2	3.11ms	38.24ms	18.77ms	15.66ms
3	3.17ms	38.36ms	18.80ms	15.66ms
4	3.11ms	38.30ms	18.86ms	15.69ms
5	3.17ms	231.45ms	133.88ms	100.22ms
6	3.11ms	38.24ms	18.77ms	15.63ms
7	3.11ms	38.30ms	18.83ms	15.66ms
8	3.14ms	38.33ms	18.86ms	15.78ms

Table II
READING OF $2048byte$ IN 8 BLOCKS OF $256byte$.

B. cfs_read() Performance

In this second experiment the file created in the previous experiment ($2048byte$ file size) is read using a sequence of `cfs_read()` calls. With each `cfs_read()` call, S_R bytes are read from the file system ($S_R \in \{16, 32, 64, 256\}$). The execution time of each `cfs_read()` call is measured. The time necessary for reading the complete file is shown for all file system modes in Figure 3.

Using CFS_APP and $S_R = 256byte$, the time to read all $2048byte$ is $25.1ms$. With CFS_MOD this time increases to $30.6ms$. With CFS_CRYPT_SW and CFS_CRYPT_HW execution times are $693.2ms$ and $381.1ms$; with CFS_CRYPT_SW_M and CFS_CRYPT_HW_M times are $608.8ms$ and $295.1ms$. Again, the overall time of reading $2048byte$ to the file is not distributed equally among the 8 separate executions of `cfs_read()` with $S_R = 256byte$ (see Table II). The first `cfs_read()` of CFS_CRYPT_SW, CFS_CRYPT_HW and CFS_CRYPT_HW_M requires a merge as this first read is performed after the previous experiment in which the file was written and the log file structure was filled. Also the 5th read requires a merge as the log file is filled. All 8 reads require reading from the flash memory followed by decryption before the decrypted data is placed in the log file structure.

When decreasing the read size S_R to $64bytes$, $32bytes$ and finally $16bytes$ the overall time necessary to read the file of $2048bytes$ increases as shown in Figure 3. However, the

times necessary for individual `cfs_read()` calls have an uneven distribution. For example, for $S_R = 16byte$ with CFS_CRYPT_HW_M the first read requires $99.8ms$ as it includes a merge, decryption of $256byte$ of data and placement of this data in the cache (the log file). The next 15 read operations require $0.5ms$ each as the decrypted data is now available in the cache. The 16th operation requires $14.8ms$ as a new block of $256byte$ is decrypted and moved to the cache.
Summary: Reading a securely stored file requires considerable more effort than reading the file form the original CFS. For example, the overall time to read a $2048byte$ file in $256byte$ blocks with CFS_CRYPT_HW_M increases by a factor of 11.7. However, not every read operation is equally expensive. For example, when using a read size of $S_R = 16byte$ with CFS_CRYPT_HW_M, read operations increase by a factor of 1.3; only when merge and/or decryption operations are necessary read operations are much more costly.

C. Cache Performance

Instead of using Codo which enables caching of unencrypted data one could use a simple solution (CFS_SIMPLE) which encrypts/decrypts data before calling `cfs_write()`/`cfs_read()` of the original Contiki file system. CFS_SIMPLE would only be usable if it is ensured that data is accessed in whole blocks that can be encrypted/decrypted in full. Hence, CFS_SIMPLE is only useful to provide a baseline for comparison here but it is not a practically usable alternative.
Writing $256bytes$ of data using CFS_SIMPLE_HW takes $11.7ms$ ($9.4ms$ for hardware supported encryption and $2.3ms$ for writing to flash memory) when writing to a file that has not been modified yet and hence the log file structure is not in use. In comparison, CFS_CRYPT_HW_M requires only $1.4ms$ as the data is written to the cache in RAM. A performance penalty only occurs for writes when the log file structure is full and a merge has to be performed (see previous paragraphs).
We note a similar performance difference for read operations. CFS_SIMPLE_HW requires $12.5ms$ while CFS_CRYPT_HW_M requires only $1.4ms$ if the data is found in the cache structure.
The performance difference between CFS_SIMPLE_HW and CFS_CRYPT_HW_M diminishes when handling smaller amounts of data in each read and write operation. This is due to the fact that then encryption/decryption times are comparable to times necessary for flash read/write operations. For example, when writing $16bytes$ CFS_SIMPLE_HW requires $1.2ms$ while CFS_CRYPT_HW_M takes $0.6ms$.
Summary: The caching functionality provides a performance benefit for individual read and write operations. Sensor network applications that access files sequentially (e.g. writing a continuous log file) may benefit from the increased read/write speed. However, applications that benefit most from the cache functionality are applications that access the same data in a file multiple times. For example, some applications may record sensor data and then perform periodically complex data processing which requires multiple reads of the previously recorded data.

VI. Conclusion

Codo is a novel framework for confidential data storage on sensor nodes. We have described and evaluated a Codo implementation for Contiki. As described, Codo addresses a number of shortcomings in existing secure storage solutions. Codo matches hardware capabilities with security requirements; in-network processing capabilities are preserved while providing confidentiality.
Security requires a processing overhead which is considerable. This processing overhead is proportional to the increase in energy consumption of a node as the CPU is active for longer. However, the CPU is generally the smallest energy consumer (radio and sensors consume far more energy) and the overall increase in energy consumption is reasonable for better security. The exact reduction of node lifetime depends on the particular CPU type and sensor platform.

VII. Acknowledgements

This work is partially supported by CONET, the Cooperating Objects Network of Excellence.

References

[1] C. Sreenan, J. S. Silva, L. Wolf, R. Eiras, T. Voigt, U. Roedig, V. Vassiliou, and G. Hackenbroich, "Performance control in wireless sensor networks: the ginseng project - [Global communications news letter]," *Communications Magazine*, vol. 47, no. 8, Aug. 2009.
[2] N. Bhatnagar and E. L. Miller, "Designing a secure reliable file system for sensor networks," in *Proceedings of the 2007 ACM workshop on Storage security and survivability*, ser. StorageSS '07. New York, NY, USA: ACM, 2007, pp. 19–24.
[3] W. Ren, Y. Ren, and H. Zhang, "Hybrids: A scheme for secure distributed data storage in wsns," in *Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference on*, vol. 2, dec. 2008, pp. 318 –323.
[4] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, nov. 2004, pp. 455 – 462.
[5] N. Tsiftes, A. Dunkels, H. Zhitao, and T. Voigt, "Enabling large-scale storage in sensor networks with the coffee file system," in *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, ser. IPSN '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 349–360.
[6] R. Di Pietro, D. Ma, C. Soriente, and G. Tsudik, "Posh: Proactive co-operative self-healing in unattended wireless sensor networks," in *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*, oct. 2008, pp. 185 –194.
[7] J. Girao, D. Westhoff, E. Mykletun, and T. Araki, "Tinypeds: Tiny persistent encrypted data storage in asynchronous wireless sensor networks," *Ad Hoc Netw.*, vol. 5, pp. 1073–1089, September 2007.
[8] "Ironkey." [Online]. Available: https://www.ironkey.com/
[9] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab, "Nanoecc: testing the limits of elliptic curve cryptography in sensor networks," in *Proceedings of the 5th European conference on Wireless sensor networks*, ser. EWSN'08, 2008, pp. 305–320.
[10] W. Hu, P. Corke, W. C. Shih, and L. Overs, "secfleck: A public key technology platform for wireless sensor networks," in *Proceedings of the 6th European Conference on Wireless Sensor Networks*, ser. EWSN '09, 2009, pp. 296–311.
[11] C. Kaufman, P. Hoffman, Y. Nir, and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)," RFC 5996, Internet Engineering Task Force, 2010.
[12] "Miracl - multiprecision integer and rational arithmetic c/c++ library." [Online]. Available: http://certivox.jira.com/wiki/display/MIRACLPUBLIC/Home