Arduino Secure File System

Ricardo Manuel Teixeira Pires

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Ricardo Jorge Fernandes Chaves

Examination Committee

Chairperson: Prof. Francisco António Chaves Saraiva de Melo
Supervisor: Prof. Ricardo Jorge Fernandes Chaves
Member of the Committee: Prof. Alberto Manuel Ramos da Cunha

September 2020

Acknowledgments

I would like to thank my family for making it possible for me to enrol in this course and allowing me to attain higher education. For encouraging and supporting me throughout this journey.

I would also like to acknowledge my dissertation supervisor Prof. Ricardo Chaves for his insight, support and sharing of knowledge that has made this Thesis possible.

A special thanks to Prof. Rui Santos Cruz who developed the template that was used for this thesis.

Last but not least, to all my friends and colleagues, especially Engineer Luís Gonçalves and Engineer Miguel Cruz for helping me with their brilliant suggestions.

To each and every one of you – Thank you.

Abstract

Nowadays portable flash devices like USB drives and SD cards are used globally. They are used to carry personal or corporate data without any protection. These devices are easy to get lost or stolen. Thus, an attacker can easily access the potential sensitive data stored in them. This study aims to tackle this problem, with the use of an external micro-controller to handle security operations and data management. We defined the system's architecture and how each of its components benefits from each approach and algorithms analyzed. From the initial prototype we developed further tuning and improvements in terms of functionality and performance, to abide by the requirements and function within our hardware restrictions. The performance results show the compromise and inverse relation between the amount of security operations performed and the system's performance, further accentuated by the tight hardware restrictions. In conclusion, this work is a significant step towards achieving digital security in flash based devices with just the use of software and cheap hardware. This work offers a solution with more digital security robustness than market alternatives which provide no integrity or authenticity while charging steep prices for limited amounts of storage.

Keywords

Flash memory; File System; SD Card; Cryptography; Micro-controller;

Resumo

Hoje em dia dispositivos de memória ”flash” como pens USB ou cartões SD são usados globalmente. São utilizados para guardar informação pessoal ou corporativa sem protecção. É fácil perder ou furtar estes dispositivos. Assim um atacante consegue facilmente aceder a informação sensível neles contida. Este estudo visa solucionar este problema com o recurso a um micro-controlador externo que irá realizar as operações de segurança digital e de gestão de memória. Foi definida a arquitectura do sistema, tendo em conta os benefícios de diferentes abordagens, e algoritmos ou sistemas analisados previamente. A partir do protótipo inicial foram desenvolvidas melhorias em termos de funcionalidade e desempenho, tendo em conta as restrições físicas do projeto. Os resultados de desempenho mostram o compromisso entre segurança e desempenho que é acentuado pelas restrições de hardware. Em conclusão, este trabalho é um passo significativo para alcançar a segurança digital em dispositivos baseados em flash com apenas o uso de software e hardware barato. Este trabalho oferece uma solução com maior robustez do que soluções de mercado que não garantem integridade ou autenticidade e são caras.

Palavras Chave

Memória flash; Sistema de ficheiros; Cartão SD; Criptografia; Micro controlador;

Contents

List of Figures

List of Tables

Acronyms

USB	Universal Serial Bus
SD	Secure Digital
SDHC	SD High Capacity
SDXC	SD Extended Capacity
I/O	Input/Output
SPI	Serial Peripheral Interface
CRC	Cyclic Redundancy Check
NTFS	New Technology File System
ext4	Fourth Extended File System
FAT32	FAT 32 bit
exFAT	Extensible FAT
JFFS	Journaling Flash File System
YAFFS	Yet Another Flash File System
F2FS	Flash-Friendly File System
ECB	Electronic Codebook
CBC	Cipher-block chaining
CFB	Cipher feedback
AES	Advanced Encryption Standard
RSA	Rivest-Shamir-Adleman

NIST	National Institute of Standards and Technology
CPU	Central Processing Unit
RC	Rivest cipher
RAM	Random Access Memory
OCB	Offset Codebook Mode
IV	Initialisation Vector
XEX	Xor-encrypt-xor
XTS	XEX-based tweaked-codebook
HMAC	Hash based MAC
SHA	Secure Hash Algorithm
MD5	Message Digest 5
TCFS	Transparent Cryptographic File System
SFS1	Secure File System
IDA	Information Dispersal Algorithm
BMC	Bare Machine Computing
ZFS	Z File System
RAID	Redundant Array of Independent Disks
EncFS	Encrypted FS
IoT	Internet of Things
PKI	Public Key Infrastructure
SSD	Solid State Drives
LBA	Logic Block Address
FTL	Flash Translation Layer
SLC	Single Level Cell
MLC	Multi Layer Cell

GB	Gigabyte
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
GUI	Graphical User Interface
KB	Kilobyte
WAF	Write Amplification Factor
3DES	Triple DES
ROM	Read-only Memory
CTR	Counter
MBR	Master Boot Record
ZIA	Zero-Interaction Authentication
OFB	Output feedback

1 Introduction

Contents

In today's world portable devices likes universal serial bus Universal Serial Bus (USB) drives or Secure Digital (SD) cards are used globally by millions of people. These devices offer a good choice for storage since they can hold several gigabytes of data. The storage capacity of these devices keeps increasing every year, as such its expected that they remain popular. At the same time, they are very small in size which gives them great usability since they can transport large amounts of information, while providing great comfort for its users.

However there are some problems with this kind of devices. They provide no digital security whatsoever. All the information is stored without any digital protection, so anyone who has possession of the device can access all of its data. Since these devices have small dimensions, they are easy to lose or to get stolen.

The objective of this project is to create a secure portable storage device with the use of a microprocessor. The idea is to implement a software solution on a micro-controller, so that all the files that are stored in the portable storage will be secure against any attacks in case it gets lost.

The general idea is for a user to be able to transfer files from his computer to a portable storage like an SD card, for example. But before being stored in the SD card, the files go through the microprocessor which handles all the security related operations, and only then are the files stored.

1.1 Objectives

The main objective of this project is to create a secure solution for small flash based portable devices. To accomplish this objective it is necessary to guarantee certain aspects. First is that the solution must be software based. The software has to be executed on a microprocessor.

In case this device is lost or stolen by a malicious individual it should be impervious to any attacks. The software solution must guarantee confidentiality, integrity, authenticity, freshness and secure deletion of data. The attacker cannot be allowed to read the contents of the SD card, modify the contents of the files to display other information, forge messages or files in the name of the original owner nor replace the contents of the SD cards with and older version, that has been deleted.

The attacker cannot be allowed to retrieve deleted files by examining the memory of the flash device. If a user deletes a file that decision must be permanent and irreversible. Even if the attacker inspects the memory of the flash device it must be impossible to discover or recover information about erased files.

1.2 High level view of the proposed solution

This solution consists in guaranteeing data security to a mobile storage device, such as a SD card, a USB pen drive or similar device. The security will be handled by a microprocessor, it can be embedded in the mobile device or separated.

This microprocessor is tasked with ensuring that the data on the device is digitally secure. The microprocessor will use a software based approach to accomplish this task. The software solution will have to make use of appropriate encryption algorithms, cryptographic hashes, and other small adjustments to guarantee freshness and secure deletion.

1.3 Structure of the Project

This document is structured in 6 chapters. The first chapter is the introduction, this chapter contains some brief explanations of the problem and the solution adopted. It also contains the objectives and some definitions to serve as guidelines for the work.

The second chapter is the background. In this chapter, different algorithms related to encryption, hash functions, cipher modes and also file systems are presented and briefly compared.

The third chapter is the related work, in this chapter a study is made of different solutions and other projects made that are relevant for the scope of this project. Some of the solutions employed by this project like key management and secure deletion were adapted from the works studied in this chapter.

The fourth chapter contains the description of the implementation of the architecture. It justifies the choices made and provides a description of the work needed to implement the solution.

The fifth chapter is where the results and their respective analysis are located.

The sixth and final chapter is where the conclusion is made.

1.4 Concept

´

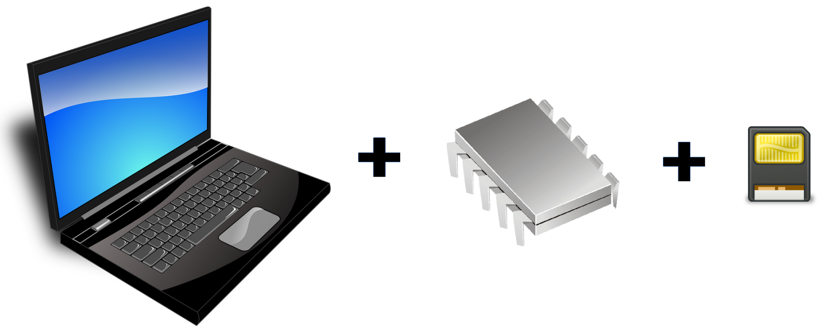


Figure 1.1: Concept diagram - Images adapted from [1] [2] [3]

The main goal of this project is to provide the user with a secure and portable storage device where he can store his files. To achieve this a microprocessor will be used. This microprocessor will be between the device the user manages his files with and the storage device. The embedded system is responsible for the management of all the data, storing files, writing or reading them from the storage device as well as encryption and other security operations, the microprocessor will be the data manager. The data in the SD card will not be able to be edited in real time, files are only capable of being edited outside of the SD card. This concept can be observed in Figure 1.1.

2

Background

Contents

The Background section features algorithms that exist today and can be used to accomplish different objectives of this project. The algorithms are analysed, pros and cons are presented and some brief comparisons are made.

2.1 File Systems

In this subsection different file systems are discussed and studied, to find out which file system is the most adequate and advantageous for accomplishing the goals of the project. The most important characteristics when choosing a file system are compatibility, wear levelling and performance.

Since this project is meant to be implemented on a micro-controller and use few resources, it only makes sense for the file system not not be a limiting factor. The SD card should be able to be used by the main operative systems like Windows, Linux and possibly Android. Wear levelling is an important metric. We are dealing with flash memory, and this type of memory tends to have a significantly shorter lifespan than hard-drives. This is why it is important to equally distribute the writes on each block of the SD card, to extend the lifespan of the memory as much as possible. Finally the performance is also an important factor. If the file system suffers from poor performance, once encryption gets added, the total time overhead can possibly discourage many people from using the file system altogether.

The New Technology File System (NTFS) is the default windows file system. It is used by windows to manage its internal drives. The advantages of using this system, are the fact that is has access control lists built-in which allows to protect file's confidentiality from unauthorised users. Built-in compression, which helps to maximise the available space. [4]

The Fourth Extended File System (ext4) is used by Linux operating systems. The system handles fragmentation better than NTFS. It has improved block allocation from ext3 to minimise file fragmentation, however it can still occur as the file system ages. [5]

Fragmentation of files leads to higher access times, since Ext4 tries to minimise this problem. It has a better performance than NTFS in this regard, especially as the file system ages and NTFS requires defragmentation processes to recover performance.

The FAT 32 bit (FAT32) file system is one of the most used file systems for flash media storage, as such compatibility is its major advantage. However this file system has a severe downfall in this project. This is due to the fact that the Arduino Library only supports the 8.3 file name convention. This limits the possibilities of encryption of the name of the file. The fact that the filename can only use 8 characters maximum compromises the confidentiality of the files' metadata. [6]

The Extensible FAT (exFAT) file system is a newer version of the FAT file system. It has several advantages over the former. It allows the media storage to be over 32 GB, has a faster storage allocation, and allows file sizes bigger than 4 GB. [4]

The Journaling Flash File System (JFFS) 2 is an updated version of the older JFFS. The JFFS system viewed the flash memory as a circular log of blocks. When the space becomes low, the garbage collector is executed. This process ensures that the file system is automatically wear levelled but at the cost of erasing the flash device too often which reduced its lifetime. This file system has slow mount times. In JFFS2 there are block lists to ensure wear levelling. This file system also has improved performance achieved with compression. [7] The wear levelling is the main advantage of this system which helps to prolong the flash memory lifespan.

The Yet Another Flash File System (YAFFS) contrasts with JFFS since it features really short mount times. It marks blocks with sequence numbers so that the valid inodes can be quickly identified at mount time. This is the main advantage of YAFFS over all the other file systems. [8]

The Flash-Friendly File System (F2FS) is a new Linux file system created by Samsung and designed specifically to perform well on flash storage devices, by increasing performance and lifetime. In the research conducted it is shown to outperform the Ext4 file system. To achieve this, the design had some key aspects to it. It avoids unnecessary and costly data copying by matching the on-disk data structures to the NAND flash memory layout. It has a cost-effective index structure. Supports multi-head logging and adaptive logging which turns random writes into sequential ones or uses multi-threaded writing in high storage utilisation. [9]

Flash memory has a drawback called the Write Amplification Factor (WAF). In short this happens when a page of memory has to be modified, at the physical level the entire block has to be re-written. If one page has 8 Kilobyte (KB) of memory and a block 64KB of memory, than the WAF is 8. In other words it takes 8 times more writes than what the file system sees at a virtual level. If poor optimisation algorithms are used to minimise the WAF then the SD card may have a significantly shorter life cycle, since the memory will wear down much faster. [10]

2.2 Symmetric Encryption Algorithms

In this subsection a study and comparison of different encryption algorithms is conducted. The important metrics for evaluating these algorithms are the strength of their cryptography and their performance in limited resource environments like microprocessors, smartcards and others. It is not enough for an algorithm to be criptographically secure, it must also conduct these operations with enough speed so that the whole system is fast and easy to use.

We focused on symmetric algorithms since these are more efficient, and designed to encrypt large files. Symmetric encryption uses one key to encrypt a file, this key is the same used in encryption and decryption. The algorithms usually cipher blocks of 128 bits which means they must be combined with a cipher mode algorithm. Most are compatible with the most common ones like Electronic Codebook

(ECB), Cipher-block chaining (CBC), Cipher feedback (CFB) and others.

Asymmetric encryption has some disadvantages like the fact that it can only encrypt plain-texts up to 2048 bits, in the case of the Rivest-Shamir-Adleman (RSA). It would be necessary to construct a cipher-block mechanism to overcome this limitation. Keys are also larger, National Institute of Standards and Technology (NIST) recommends keys of at least 1024 bits, which is 8 times the necessary size for symmetric encryption algorithms like Advanced Encryption Standard (AES) also known as Rijndael. [11] Since users can have files that have gigabytes in size, it is preferable to use symmetric encryption which can handle these sizes.

2.2.1 Algorithm Analysis

We will focus mainly on algorithms that are proven to be safe, and NIST approved. There are many encryption algorithms that claim to be secure but are not verified by any reliable sources, so caution must be exercised when considering an encryption algorithm since represent a crucial part of digital security.

The first algorithm to consider is Triple DES (3DES). DES was a very fast and secure algorithm for its time, but now has become vulnerable due to the increase of Central Processing Unit (CPU) processing power. Triple DES was then designed to take advantage of DES strengths and still provide a secure encryption. There are two ways of using DES to use three different keys and encrypt with every single one of them, or to use two keys and perform a sequence of encrypt with key one, decrypt with key two and encrypt again with key one. The first technique is vulnerable to key related attacks as demonstrated in [12], and Tripe DES with only two keys also has also been considered a weak cipher by NIST. [13]

The NIST and industry approved encryption algorithm is AES or Rijndael. Rijndael was the candidate chosen to become the Advanced Encryption Standard since it was the best overall algorithm. It offered great security while also achieving an excellent trade-of with performance, since it had the best performance overall. However it is still important to check if Rijndael still has the best performance in microprocessors, or if another of the finalists has the edge in this field. [14]

The Serpent algorithm claims to have the best performance in hardware implementations. It also claims to be possible to implement the algorithm using only 80 bytes of random access memory (RAM) [15]. This gives Serpent a significant lead in terms of space usage, although the performance must be compared in software implementations rather than only hardware.

Rivest cipher (RC) 6 is the improved version of RC5. It tries to use operations that are already efficiently implemented in processors, like rotations or integer multiplications, to encrypt data. Its design is well suited for microprocessors, but there may be cases where it is preferable to just use an integrated circuit dedicated to RC6. [16]

The MARS algorithm focused mainly on achieving a trade-off between security and performance.

The algorithm has high performance in software implementations, since it was designed to take advantage of operations available in computers. [17]

2.2.2 Performance analysis in smartcards

All the algorithms presented claim to have good performance in either microprocessors, software implementations or both. A clear winner is not evident just from analysing their proposals. It is necessary to use a real implementation of the algorithms and test them against each other. In this benchmark, smartcards were used to measure the performance of the algorithms since they have extremely limited resources. This is similar to the scenario of a microprocessor where Random Access Memory (RAM) and processing power are limited. So it is safe to infer that if an algorithm performs well in a smartcard, it will also have a good performance in a microprocessor. Important performance metrics are the amount RAM and Read-only Memory (ROM) used, since these are scarce resources and also the encryption time.

According to a study conducted by Toshiba Technology Centre [18] to evaluate the performance of the different AES candidates on a smart-card using software based implementations in C language, it is clear that the Rijndael algorithm, commonly known as AES, has the best performance in smartcards using a software implementation, as seen in Table 2.1. It can be implemented with extremely limited resources since it only consumed 66 bytes of RAM and required 980 bytes for storage. It is also the fastest algorithm by far. The MARS algorithm required a complex and difficult implementation to work on a smart-card, due to its complex structure. MARS also had other disadvantages of being implemented on a smartcard that causes the algorithm to be vulnerable to timing attacks.

Cipher	RAM (bytes)	ROM (bytes)	Encryption Time (clock cycles)
MARS	572	5468	45588
RC6	156	1060	138851
Rijndael (AES)	66	980	25494
Serpent	164	3937	71924
Twofish	90	3937	71924

Table 2.1: AES Smartcard Benchmarks - Adapted from [18]

2.3 Stream Ciphers

Stream ciphers are functions that behave similarly to one time pads. Where the pad is computed from a combination of a random vector or carry value and XORed with the plain-text. They usually have the advantage of being faster than block ciphers and using less memory since they work with only few bits at a time. They are normally used with continuous streams of information, this is a similar case to the problem presented since the microprocessor will not receive the files all at once but in continuous chunks of data. Stream ciphers have the disadvantage of requiring total re-encryption each time a file is updated. Parallel processing is not possible with stream ciphers.

In CFB an Initialisation Vector (IV) is used. First the IV is ciphered with the Key, then the resulting cipher-text is XORed with the n most significant bits. The resulting cipher-text is used in place of the IV for the next n bits and the process repeats. The Output feedback (OFB) functions similarly to CFB but the carry value results from the computation of the IV with the encryption algorithm instead of the cipher-text. Counter (CTR) mode is another stream cipher. It functions similarly to OFB, but instead of using an IV it uses a counter that produces random values. The sequence of counters must guarantee that each block in the sequence is different from every other one. [19]

2.4 Block Ciphers

In this subsection different block cipher modes are presented. Block ciphers play an important role in the encryption stage. The encrypting algorithms only encrypt in small blocks or in streams. As such it is important to chose a good cipher mode to determine how the blocks relate to each other. Block ciphers are more geared towards file encryption.

In ECB mode the cipher function is applied directly and independently to each block of plain-text. The decryption process functions in a similar manner. The cipher-text is composed of multiple blocks ciphered with the key and algorithm. ECB supports parallelism. It has the disadvantage of the same plain-text generating the same cipher-text when ciphered with the same key. Useful for small string encryption only.

CBC uses an IV to combine (XOR) with the first plain-text, then it ciphers the combination and uses the resulting cipher-text as the next IV. The IV can be known, however it is important to guarantee that it has a randomised generation a different IV should be used for each encryption. Since the next block to encrypt depends on the previous one, parallelism is not possible in CBC. It is possible in the decryption process though, since all blocks are available. CBC is prone to padding oracle attacks, which try to exploit the padding that the algorithm requires. [19]

The Offset Codebook Mode (OCB) is a cipher mode that tries to provide both authenticity and confidentiality. It stands out due to the fact that it manages to achieve authenticated encryption in almost the same time as the other cipher modes. It is also much faster than encrypting a file and then authenticating it. It uses a nonce instead of an IV for the ciphering, The nonce is recommended to have 96 bits of length, and can be created using a counter. It then uses this nonce to XOR a block before and after encrypting it with the Key. The authentication is achieved with a tag that is computed by XORing the encrypted checksum of all the plain-texts with the last nonce. [20] This algorithm however is vulnerable to collision attacks. The attack is performed by waiting for a specific type of collision to occur on 128 bit values. When the collision happens, the cipher-text can be modified without modifying the authentication tag [21]. This shows that the authentication tag is useless at preventing message forgery.

The XEX-based tweaked-codebook (XTS) is a cipher block algorithm used to encrypt entire disks. It is based on Xor-encrypt-xor (XEX). XEX uses two keys. Key one is used to encrypt XORed plain-texts. The other is used to XOR plain-texts and the cipher-texts resulting from the encryption with the first key. For each block, to compute the XORs, it is necessary to compute a derived key from the second key. Differing from XEX, XTS uses cipher-text stealing to allow for sector sizes with non-divisible block sizes. It must still use two keys. The major advantage of this block cipher is the fact that it allows random writes without need to re-encrypt the entire file, which is particularly useful for log files for instance. [22]

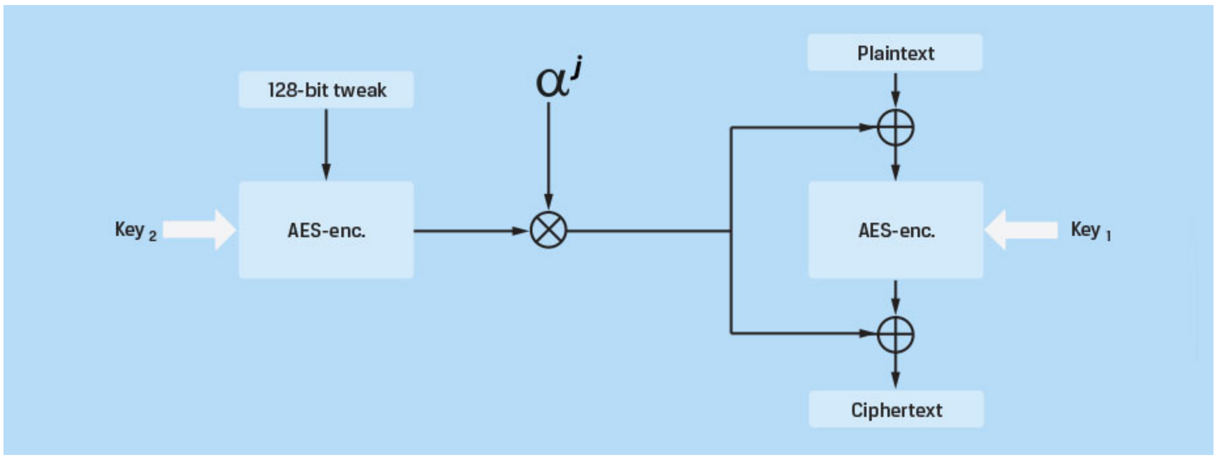


Figure 2.1: XTS cipher mode - Extracted from [23]

2.5 File Authentication and Integrity

While file encryption is extremely important to guarantee data security, authenticity and integrity also play an important role. This means that a good file hash must be employed to protect the data against file adulteration or forgery.

Cryptographic hashes rely on complex mathematical computations that are easy to compute in one way, but unfeasible (in useful time) to compute in reverse. This means that they can be stored in plain-text and be secured or encrypted as well to provide an extra layer of security.

Just like the algorithm encryption choice, it is also important to chose a fast hash function that works well in microprocessors. These hash functions are going to be hashing files that can have hundreds of megabytes, or even gigabytes in size, if the performance is not great, then the microprocessor will struggle to guarantee integrity and authenticity since this part will start dragging the upload time of files.

These hash functions will also have the task to protect the file's metadata. In this case the only metadata that needs protection is the file name. As such a strong hash must be chosen since filenames can have short names of just one character. The hash function should provide ample security in this case, and perhaps a SHA-3 algorithm would be more appropriate than reusing the same algorithm used for file authentication and integrity, as this would give the attacker 2 different cipher-texts to exploit and increase the attack vector.

2.5.1 Message Authentication Code

A simpler way to authenticate messages is by using a message authentication code (MAC). A MAC can protect the authenticity of a file and also its integrity. MACs can be generated with block-cipher encryption algorithms. For example, a concatenation of a file CRC code and the user's digital signature, encrypted with the file's secret key can be considered a MAC. Another way of verifying file's integrity is with the use of Hash based MAC (HMAC). A HMAC uses a cryptography function like Secure Hash Algorithm (SHA) 2 or SHA-3 and a secret key. When using only a hash function like SHA-2 anyone can compute a file's hash as long as they have access to the plain-text. However with a HMAC only someone that knows the secret key can compute the HMAC. This makes it much more secure against file tampering and forgery [24].

2.5.2 SHA-0 and SHA-1 Hash Functions

The SHA-0 algorithm was the hash standard in 1993. It is an old hash function which was successfully exploited in the international cryptography convention CRYPTO 98. [25] As such it is not a valid hash anymore. Message Digest 5 (MD5) was designed to be fast on 32 bit machines. The reason to mention MD5 in this report, is because it is still widely used. [26] This hashing algorithm is quite old, and vulnerable to attacks, such as differential attacks. It is not very difficult to conduct this type of attacks with modern hardware, which makes this algorithm quite vulnerable by today's standards. [27]

MD6 is another hash function, it has better cryptographic security since it can withstand differential attacks. It is an algorithm that is well optimised for parallel processing. MD6 can be adapted to work as a HMAC. [28]

2.5.3 SHA-2 Hash Functions

The SHA-2 functions are a collection of functions that produce message digests with different sizes starting from 224, 256, 384 and 512 bits. We will focus mainly on SHA-256 and SHA-512 since the others are truncated versions. SHA-256 is approved by NIST as a secure hash function.

It is a very used algorithm to provide authentication and integrity. SHA-256 uses a block size of 512 and iterates 64 rounds. SHA-512 uses a block size of 1024 bits and iterates 80 rounds. The algorithms cannot be cracked by brute force attacks in feasible time. Birthday attacks and the Poisoned Block Attack also fail against both functions. The only successful attacks made were against reduced versions of the functions. [29]

Tiger is a hash function that tries to provide the same level of security as the SHA-2 family. It was designed for 64 bit architectures. The function can withstand message modification attacks, and was also resistant against pre-imaging attacks. However, it was possible to attack Tiger with collision attacks using only 16 to 20 rounds. Since the algorithm only has 24 rounds, it shows some vulnerability against this type of attacks. To safely implement this algorithm it would be necessary to reinforce it with additional rounds to make sure that its not possible to crack it in the near future. [30]

Whirlpool is a hash function on par with SHA-2. This function is not designed to perform well on a specific platform, as such processors that have an 8 bit architecture or 64 bits, can benefit from its implementation. Whirlpool is secure against differential attacks, and attacks against the internal block cipher are also unfeasible. Whirlpool is more scallable than most modern hashing functions. It can be efficiently implemented on numerous and different platforms and also supports parallel execution. Its a good choice for smart-cards since it does not require many resources and can perform well on constrained scenarios. [31]

2.6 Hardware

For file storage we will use a SD card, like the one depicted in Figure 2.2. SD cards use flash storage to archive all of its content. They rely on the flash memory controller to decide which blocks get written by a new file. This controller is responsible for making sure that the memory gets worn out evenly, and tries to maximise its longevity.

Figure 2.2: SD card

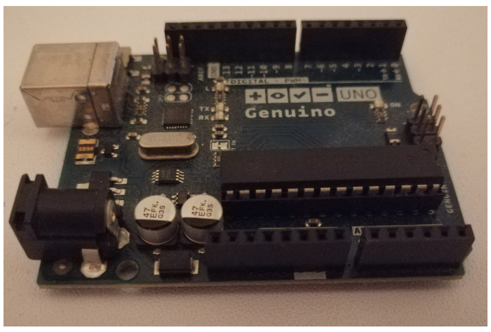


Figure 2.3: Arduino Uno

To handle the data management in the SD card and to perform security operations a micro-controller was used. An example of a micro-controller can be seen in Figure 2.3. SD cards were designed to provide security, storage and performance for audio and video devices that were emerging at the time, like digital cameras, smartphones, and other devices. It is capable of some limited security operations like blocking writing and deletion to the SD card with its locking mechanism. There are different versions of SD cards, SD High Capacity (SDHC) and SD Extended Capacity (SDXC), these acronyms are relative to the storage space which can go from 2 Gigabytes to 2 Terabytes in the case of the SDXC. It uses an Serial Peripheral Interface (SPI) for Input/Output (I/O) operations. The SD communication functions under a command and data bit streams. The command is sent from the host to the SD card via serial, to which the SD card replies with a token. Data transfer is done in blocks which are always verified with a Cyclic Redundancy Check (CRC). The read and write functions are invoked by the host and performed by the on-board controller. These operations are completely independent of the host. The SD cards uses a 512 byte sector size, smaller blocks are not allowed, to store its data. Other variants with larger storage may have block sizes larger than 512 bytes, but that is the default value for most cards. It is also possible for a host to use multiple SD cards, issuing reads and writes to different cards simultaneously. [32]

Arduino and Raspberry Pi are example of integrated boards. These boards have integrated modules of hardware and circuits, like network interfaces, power sockets, digital and analogical digital pins. They can also be expanded with other modules. For example, it is possible to add an SD card reader (called the Shield SD which can be observed in Figure 2.4) to the Arduino board so that it can read and write to the card. The Raspberry Pi can run different types of operative systems, and different programming languages. Arduino on the other hand only supports its native language which is extremely similar to C++. There are also different variants of Arduino boards like the Arduino Mega, which is larger, has more processing power as well as more RAM and flash storage for scripts. The Arduino Nano which has identical hardware specs of the Arduino Uno, but with less pins.

The SD shield is a module that can interface with the SD card. In the case of the Arduino the shield has a physical socket to connect with the SD card. The shield can then connect with an Arduino via breadboard or connecting with only pin cables. Currently with this shield, it is not possible to directly integrate it with an Arduino. In Section 4.4 we will discuss why the Arduino Uno was the hardware chosen for this project in more depth.

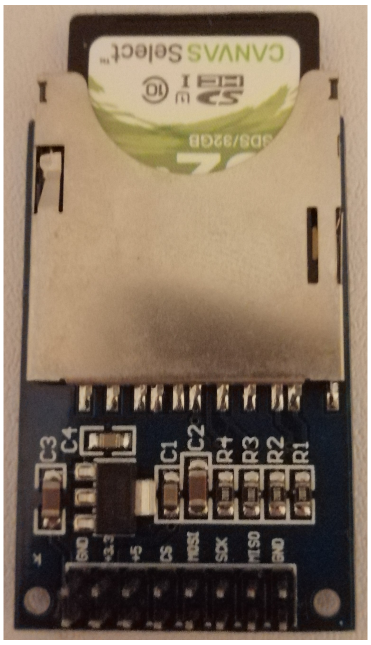


Figure 2.4: Shield SD with a SD card inserted

3 State of the Art

Contents

This section analyses solutions that are either similar to the problem presented for this project or offer partial solutions that can be adapted for this project. Some are complete solutions to the problem, others focus on guaranteeing encryption, secure deletion or just integrity for example. These works need not to focus specifically on just the topic of their subsection. Frequently they cover multiple topics. The reason for being placed in a particular subsection is either because they mainly focused on that aspect, or the work they conducted was considered of special importance for this project in that area.

3.1 Encryption

This subsection is for works that focus mainly on file systems or interacting with file systems. Some try to only add cryptography and be as independent as possible, others may try to develop new file systems methodologies to provide encryption.

The Transparent Cryptographic File System (TCFS) is a solution proposed for file transfer between a user and a remote server. The main idea is to have multiple users connect to a remote file server to access their files, since their workstations would have very limited disk space. This file system guarantees that the files are not readable by users other than the owner of the file, including the superuser of the file system. Communication between the user and the remote server is secure. [33]

Keypad is a file system designed to offer security in remote devices such as laptops, USB drives, and others. It also provides the user with an auditing mechanism to detect whether or not there have been access attempts to the files in case the device gets lost. Furthermore, in case the remote device is lost, the user still has the capability to disable future file accesses even in the absence of device connectivity. Keypad uses a similar strategy applied in the TCFS. It uses different keys for each file and stores those keys on a remote auditing server. After the transaction, the key is securely deleted. When a user wants to access a certain file, he has to download a specific key from the server. All file operations made by a user are registered in a log file. This log allows a user to audit his remote devices and check if there have been attempts to discover the password. As a last resort the user can chose to delete all random keys for the lost device which also makes the main password obsolete. [34]

In Separating key management from file system security, the Secure File System (SFS1), is a system designed to handle the issue of key management. By segregating key management from the file system, is ceases to be necessary for file names and encryption keys to be mapped to each other. To achieve this, the system uses self-certifying pathnames and separates user authentication from the file system. In order to provide security for the files it was established that SFS1 would only rely on the filenames and public key to store and encrypt the files. The location of the file is hashed and only possible to know for the users with the right keys. The key managing part of the system is done with multiple different techniques. SFS1 uses manual key distribution, certification authorities, password authentication and public key infrastructures to handle key management. [35]

A secure cloud distributed file system is a modern solution to offer security in files which are stored in cloud servers. The architecture of this system is based on the Information Dispersal Algorithm (IDA). This algorithm splits a file into n slices and stores each slice into dispersed storage nodes. It combines this slicing technique with authentication and access control lists. With these techniques and by making the number of slices sufficiently large, it claims to achieve a high robustness in terms of confidentiality. This system also requires a metadata server to store the location of each slice. The metadata itself is also sliced and then stored to avoid potential data leakage. The testing performed in the study showed that by using IDA, this system had better data availability than systems which relied in data replication. The testing conducted showed that with a sufficiently large number of slices an attacker can get access to a partial amount of data. It is not necessary for an attacker to obtain all slices for data leaks to occur. [36]

Secure USB based file system for Bare Machine Computing (BMC) applications provides confidentiality and integrity to its users while also offering ease of use. The user sees a normal drive on his computer when he interacts with this system. The security operations are all hidden from the user. The system provides a number of API functions for file manipulation, like file creation, deletion, writing and others. The system analyses the Master Boot Record (MBR) to organise a memory map of the USB device. This system is claimed to be more resistant against virus attacks, since it does not allow for strange code to be executed. [37]

In Design of USB Storage Encryption Device Based on AES-XTS the AES algorithm with XTS cipher mode is used to grant confidentiality to the data in the USB. In this system the key size is 128 bits. It uses a physical device to perform the encryption and decryption of data. This device contains a microprocessor to perform the tasks, and adapters to communicate with a computer and a USB drive. In this system the computer first sends the data to the physical device which encrypts it and then writes the encrypted data in the USB drive. It also has an authentication mechanism to verify if a user has read and write permissions. The implementation of the XTS cipher has been tweaked for better single-core performance since XTS is usually implemented using two cores for maximum efficiency. In this implementation there is a selector data decided when each key is used and whether to XOR the cipher-text with the second key or to encrypt the data, this scheme is better illustrated in Figure 3.1. In this circuit there are two selectors, one which decides which key is used, the other for selecting the data or the XTS tweak. First Key2 and *h* (which represents the tweak used in XTS) are selected and the encryption is performed in the AES encryption/decryption module. The result of the tweak is XORed with the data before it is encrypted with Key1. After the encryption the resulting cipher-text is XORed again with the encrypted tweak. Basically this model allows XTS to function continuously in a sequential way. [38]

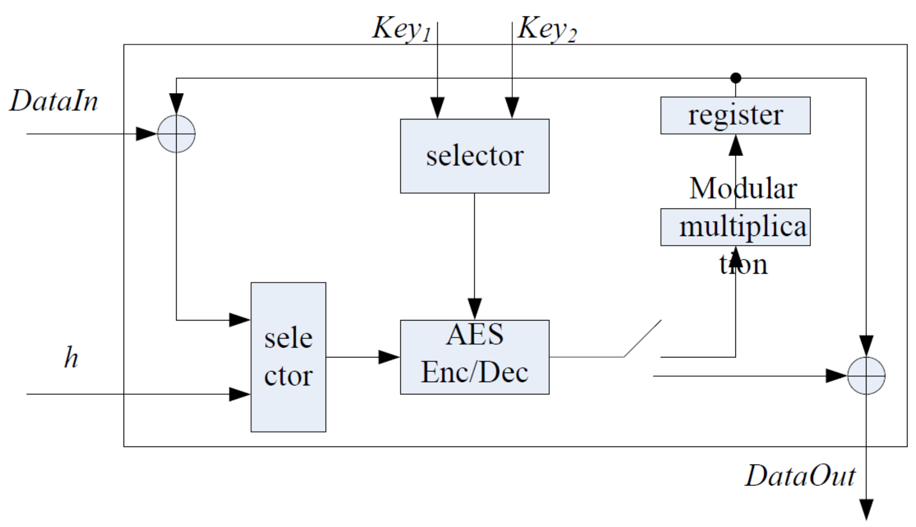


Figure 3.1: AES-XTS modified - Extracted from - [38]

3.2 Integrity

The focus on this subsection are systems that try to guarantee integrity of their files to the users.

The Z File System (ZFS) is modern file system that tries to ensure file integrity by managing memory and disk faults using numerous reliability mechanisms, as claimed by its developers. The data corruption can originate from either software or hardware faults. It provides data integrity, transactional consistency, scalability but also some other useful features like snapshots and copy-on-write clones. [39]

ZFS integrity mechanisms include checksums, replication, storage pools. For data integrity ZFS relies on checksums. The checksums are kept separately from the corresponding blocks and whenever ZFS uses a block it first checks it against its checksum. This allows the file system to detect silent file corruption. For data recovery ZFS uses Redundant Array of Independent Disks (RAID) techniques but also keeps replicas of important on-disk blocks. Both the metadata and data of the file are replicated. It also uses storage pools that can be either virtual or physical devices in order to increase the reliability by using a similar configuration to RAID. The file system includes automatic repairs in mirrored configurations and uses disk scrubbing to detect latent sector errors. ZFS has two main file disk structures. Objects which encompass all blocks in the disk or block pointers which are used for checksums processes. ZFS was tested for on-disk data integrity, on-memory data integrity. In the case of on-disk integrity the file system can detect single and multiple block corruptions on either data and metadata. However it can't recover from multiple block corruptions. ZFS is vulnerable against memory corruptions since once the blocks are written to memory they've already been submitted to a checksum and thus are assumed to be correct. The system does not have a mechanism to defend against memory corruptions effectively. [40]

In Integrity Checking in Cryptographic File two solutions are presented for this problem. The design of the solutions uses two characteristics of file systems to its advantage, low entropy of file contents and high sequentially of file block writes. The solution also attempts to include the use of Merkle trees, defence against replay attacks of previously overwritten blocks. Merkle trees are structures that are used to authenticate data with constant-size trusted storage. [41]

The algorithms used were implemented in the Encrypted FS (EncFS), an open-source cryptographic file system, and were tested with AES and Blowfish in CBC mode. The first algorithm uses the observation from *"Space-Efficient Block Storage Integrity."* [42] that the integrity of the blocks that are efficiently distinguishable from random blocks can be checked with a randomness test. The second algorithm is based on the fact that many workloads feature low entropy files. The blocks are stored before encryption and if the compression of the block is high enough then the hash of the block can be stored in the block itself. In the tests conducted, there is no clear winner since each algorithm works best in a particular workload. The recommendation is that all are implemented in a cryptographic file system and chose the best suited algorithm for the case. [41]

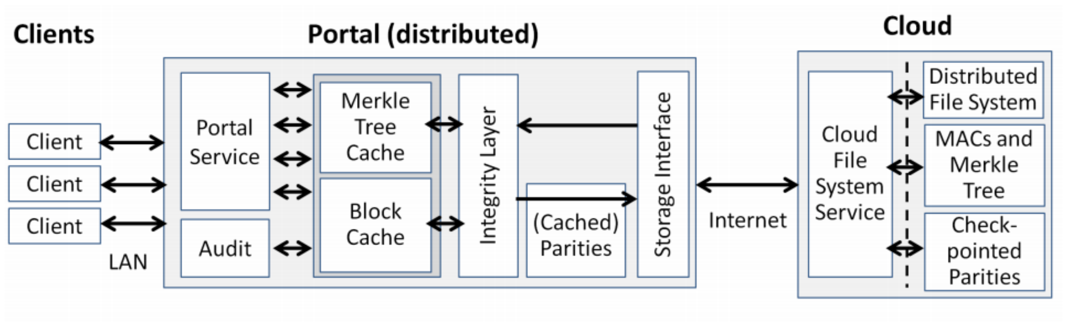


Figure 3.2: Iris system architecture - Extracted from [43]

Iris is a file system created to support workloads of large enterprises while offering authentication, file integrity and freshness. It is a file system that is resilient against untrusted environments such as cloud servers. This file system requires a lot of caching to be made on the client side, to minimise latency and for the operations made by the Iris server to be efficient enough, so that they don't restrict usability to the users. The system resides in between the client and the cloud. A network of distributed "portals", like the one depicted in Figure 3.2, are responsible for managing the inbound and outbound data from the company. These portals are composed of different modules, different types of cache, audit modules and others. These portals store in cache data and metadata recently accessed by users, they also check the integrity and freshness of all files. The audit module sends periodic challenges to the cloud server to verify the availability of the entire file system. It also uses its cache services to help recover from data corruptions. For authentication Iris uses two layers. In the lower layer it stores a MAC on each file block, every time a user writes to the file system. The authentication of each block file, instead of the whole allows random accesses to the files. For freshness the system also authenticates the version of each block. This prevents rollback attacks where users are presented with an earlier state of the file. The upper layer is responsible for ensuring the integrity of the version counters. This layer uses a Merkle Tree structure to ensure the integrity of the files and efficiency of operations. [43]

S-Audit is designed to be an efficient data integrity verification for cloud storage environments. This system is designed to withstand a scenario where an attacker has full permissions over the file system to write and delete files as he pleases. The attacker's fingerprints are also unknown to the system. The communication between user and server is assumed to be authenticated and secure. With this system users can verify the integrity of a remote file by selecting a random chunk of information by downloading the proofs, that are small and have constant size, which compress the data and signatures. In the tests done it was shown that S-Audit requires marginal numbers of bandwidth but requires more storage size when compared to RSA. The cost of signing a file increases significantly when files start to get large, meanwhile RSA has a constant signing time. This incurs in a heavy overhead for the user if he wishes to upload large files to the cloud. [44]

3.3 Authentication

This subsection is for works related to user authentication and secure communication.

Zero-Interaction Authentication (ZIA) is a system designed to provide file encryption without compromising performance or usability. It is designed for portable devices such as laptops. It uses a token which is responsible for providing the master key which encrypts all the other keys. This token is a physical object that is carried by the user, losing the token is a serious loss that may compromise the use of the laptop. The token has limited storage and processing capabilities which means that all the file encrypting keys have to be stored on the laptop. This system enforces two requirements. The token cannot be shared with other users. The communication between the token and the laptop must be made via an authenticated and encrypted wireless link. This system encrypts the entire hard disk of the laptop. For performance reasons when the user is logged in and working on the laptop the cached files are not encrypted. Items in the cache are only encrypted when the token leaves the range of the laptop. The acquisition of file encrypting keys is reliant on a local administrative authority that is responsible for distributing keys and holding them in escrow. To prevent attackers from knowing when a key is being acquired it encrypts file in the same directory with the same key, this minimises the number of times a key is requested. [45]

In Decentralised User Authentication in a Global File System the SFS2, was designed to provide confidentiality, integrity and server authenticity. This system relies on asymmetric cryptography. Each server has a private key which is undisclosed, and only disclose their public keys. To establish authenticated and secure connections clients use the public keys of the servers. These servers are self authenticated. When a client wishes to authenticate himself, he first signs an authentication request with his own private key, which is then sent to the authentication server. The authentication server verifies the signature and issues credentials for the user, the credentials are also sent to the file server. It is possible for the authentication server to share the same machine of the file server, or to run completely separated. [46]

A Lightweight Identity-Based Authentication Protocol is a system that is designed to be lightweight and providing a secure authentication method to be used in wireless networks. In the client a dedicated encryption chip is used to store the user's information and the shared key pool. The server also has an encryption embedded card that is used for the token generation algorithm and to store the storage key. The authentication protocol starts by generating an authentication token. This token is computed in both the client and the server. The token is generated in the encryption card, which computes it by using the prestored key pool. Then the client sends it's token to the server to be validated, if it is successfully validated the server sends its code to be validated as well. The tokens also include time-stamps. [47]

In Authentication of Internet of Things (IoT) Device and IoT Server using Secure Vaults the authors designed a system that provides authentication with resort to secure vaults. These secure vaults store n keys of m bits in size. During the deployment phase the secure vault is shared with the server and the client, they are then stored encrypted in both of them. The authentication is then carried out by using a challenge-response mechanism. The challenge begins with a request made by the client, which sends its unique id and session id to the server, this is sent in plain-text. If the server confirms that the ID received is valid it sends the challenge to the client, also in plain-text. The client has to compute an encryption key based on the challenge, if done correctly it then sends a challenge to the server, and this message is sent encrypted with the newly created key. After the server solves the challenge it sends the response to the client. If both parties solved each challenge successfully they then negotiate a session key. After the session ends, an HMAC is computed based on the exchange of contents, this HMAC is used in the computation to generate a new value for the secure vault. [48]

Arduino Based Smart Fingerprint Authentication is a system that uses an Arduino board and fingerprints to authenticate users. This system uses an Arduino Mega 2560 board, a touchscreen, a fingerprint sensor, a sound alarm, and a module to send messages to mobile phones. A user registers himself in the system by entering his fingerprint and phone number, then a photo is given to him as a password for backup. If a user chooses to login to the system, he needs to interact with the touchscreen and has to select whether or not he is a registered user. If registered, then the fingerprint must be presented to the sensor to be tested. If the fingerprint is recognised either the system sends a one time password to the mobile phone of the user or he must select his image password from a pool of images that are presented to him. If the user enters the password incorrectly three times, the alarm will sound. In case the user is not enrolled in the system then the image or password request is followed, so in this case the system administrator is informed and the image or one time passwords are sent to him instead of the user. This prompts the unauthorised user to communicate with the system administrator so access to the device can be granted. [49]

3.4 Secure Deletion

This subsection talks about multiple projects related to secure deletion and can provide insight on what is the best or more efficient way to offer secure deletion of files to a user.

In Secure Deletion with Ephemeral Keys the objective is to support high availability of data until the moment of deletion. After deletion it should be impossible to read or recover the deleted data. The design of this system allows for two modes, one where the deletion occurs after a known and established file expiration time. The other is deletion on-demand. The system guarantees certain properties like, allowing an authorised person to access the entirety of the records from the file system. Unauthorised users cannot access system files. No one can read data which has expired. The system is designed as follows, there is a service that provides ephemeral keys to users, this service is called the “Ephemerizer”. The ephemerizer has a long term key, as do the users. These keys are distributed by a Public Key Infrastructure (PKI). The ephemeral keys are never copied or shared across users. To communicate with the Ephemerizer the user sends a combination of the secret key encrypted with an ephemeral public key, and a message encrypted with the public key, and an identifier of the public key. This combination is then obfuscated with a function called the Blindable function. The server then identifies which private key to use based on the identifier and unencrypts the secret key, and applied the Blindable function to the key, then sends it back to the user which applies and inverse function to retrieve the key. Each file is encrypted with a different and unique key to ensure on-demand deletion. The system manages a table which maps each key in RAM to its file. Then copies this table to remote storage to build back-ups. When a user wants a file to be deleted, the ephemerizer deletes the file’s secret key, making it unrecoverable, and re-encrypts the table with a new key. [50]

PurgeFS is a file system designed to make sure that deletion of data is permanent and is impossible to recover or restore it. It was designed to be able to be integrated with any running Operative System. This system overwrites file data and metadata more than once using a number of patterns, like the ones approved by NIST. PurgeFS can erase data in asynchronously or synchronously. In synchronous mode it will wait for all the overwrite operations to finish, in asynchronous mode it will create a map of the files to be overwritten. It can also eliminate only sections of a file. The system is highly portable since it was designed as a file system extension and can easily be added to most existing file systems. It performs data purging almost immediately after a file’s deletion. In non-workload environments it is shown to be non-intrusive and quite efficient due to its asynchronous mode. [51]

In Secure Deletion in Flash Memory the method employed takes into consideration that flash memory cannot be handled in the same way that hard-drives are since flash memory can only be re-written so many times before the sectors become unusable. Because of this, frequent memory overwriting approaches can not be used, or they can seriously harm the lifespan of the devices. This study uses encryption to guarantee secure deletion. First a key is created to encrypt data. Before the file is saved it is encrypted with the file key, then afterwards the file key is saved in the header section. When a user tries to remove a particular file, the solution will only overwrite or erase the header block where the key to the file is stored. The key is created through the usage of several key generation functions. It can be generated using random or pseudo-random number generators or by using a pass-phrase and a key generation algorithm. This is to ensure that the key generation is sufficiently random to be unpredictable and cryptographically secure. The pass-phrases can also be hashed to prevent guessing and brute-forcing attacks. The Master key is based on the password of the user, and is calculated using a hash function. They suggest using this hashing function at least a thousand times to prevent password attacks. All other random file encryption keys are encrypted using the master key, and each file is encrypted with its own unique key. When a user wants to delete a file, the header block is searched for that key and the key is overwritten with an unclassified pattern. [52]

In Reliably Erasing Data from Flash-Based Solid State Drives (SSD) there was a study conducted to create efficient ways to securely delete data from SSDs, but with minimal wear to the flash memory. Due to the fact that SSDs use flash memory, they cannot rely on the same methods that hard-drives use, since this would cut their lifespan significantly. This study compares different solutions and how well they perform when deleting data. The study analysed three different methods to sanitize an entire drive, the first is to use the built-in sanitization commands. These commands write ones and zeros to erase data. Of the tested SSDs some did not have these commands, and several others could not successfully execute them either because they either failed or the data had not been erased properly. This means that this technique is not a reliable way to securely delete data. Another technique is to overwrite data. This technique is quite popular and was used in some of the previous studied works. Due to the fact that SSDs may use compression when storing data, it is important to use randomised data to perform the overwrites. This is to make sure that the data gets completely written and is not compressed. In the tests conducted it was found that in most cases overwriting a disk twice was enough to sanitise it. The main problem of this method is that its performance was very slow for a single disk overwrite. Degaussing is a method for destroying hard-drives, it is fast and effective but not very relevant for this project since it is important that the SD card remains usable. Another way is to encrypt the information before it is transferred to the disk. Only the memory section that contains the file key is sanitized when a file is deleted. This method is very quick because you only need to purge a small amount of data. It has several drawbacks, the keys need to be properly sanitized and flash devices do not have any of those functions or commands built-in. All these software-based methods have limitations, modifications to the flash translation layer are required to overcome this problems. This strategy does however make recovery from file corruption difficult and does not extend to standard SSDs. This approach makes changes in the Flash Translation Layer (FTL) so that when a file is deleted it scrubs the Logic Block Address (LBA) by setting everything to 0. This requires caution since the LBA may be shared with an active file. This approach is not supported by all manufacturers and requires that the device is tested to see if it is indeed possible to perform LBA scrubbing. The next step is the configuration of the FTL to enable it to use file scrubbing. Three methods were used. Immediate scrubbing that blocks all writing until the scrubbing is complete. This is considered the safest method and has little impact on performance since it is possible to execute the program in parallel and to scrubb it. Background scrubbing offers better performance because it will wait for the writing to finish before proceeding with the scrubbing. Scan-based scrubbing has no performance penalty on normal write operations. In terms of performance, immediate and background scrubbing the cost of scrubbing varies, in Single Level Cell (SLC) SSDs the cost of scrubbing is negligible since it can execute in parallel with normal writing. In Multi Layer Cell (MLC) based SSDs the cost can be very high if the device only supports a small number of scrubs before an erase. This scrubbing method does have a cost, which is a reduced longevity of flash devices. It can also cause errors to start appearing. [53]

User Level Secure Deletion (USRM) provides secure deletion at a user level with an innovative technique. The process starts with the creation of a temporary file, the file size is then increased until it fills all the empty space left in the disk. This filling of free space is done to speed up the overwriting process. This process is executed using the function ”fallocate” which is present in the Linux kernel. Not all file systems support this function, such as the FAT file system. The study used an alternative method for these cases, which consists in creating a temporary file, then moving the file pointer to the end of the file and writing a 0, this is enough for the system to consider the free space as full. Afterwards the information of the target file, to be deleted, is removed. A junk file is then created with random information to overwrite all the space previously occupied by the target file. In the tests conducted it was shown that the target data to be securely deleted was successfully overwritten with random data at the physical layer. USRM also has improved performance when compared to other functions such as ”sfill” and ”srm”. [54]

3.5 Similar Solutions

This subsection is dedicated to complete solutions that try to tackle the same problem as presented in this report. A summary and comments are then made about these solutions and how they can be adapted to the project.

A Secure Flash Card Solution for Remote Access is one solution to the problem of guaranteeing security in a SD card. This solution relies on two main components, a tamper resistant module embedded in the SD card, and a server for authentication. The tamper resistant module contains user authentication information and the file keys, its modules can be observed in Figure 3.3. The communication between the user and the SD card is also encrypted, the communication protocol can be seen in Figure 3.4. [55]

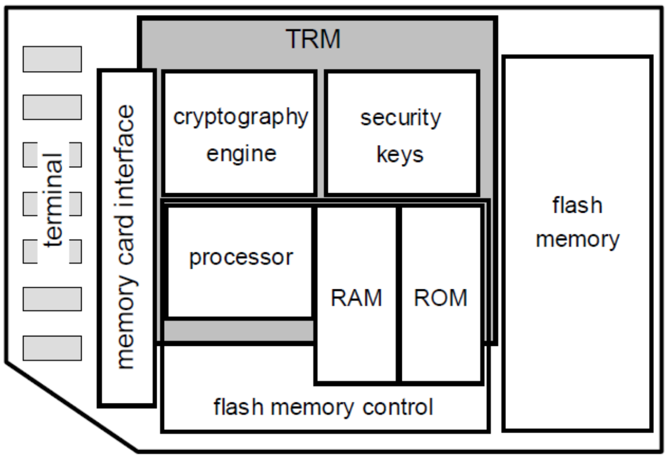


Figure 3.3: SD card with embedded security - Extracted from [55]

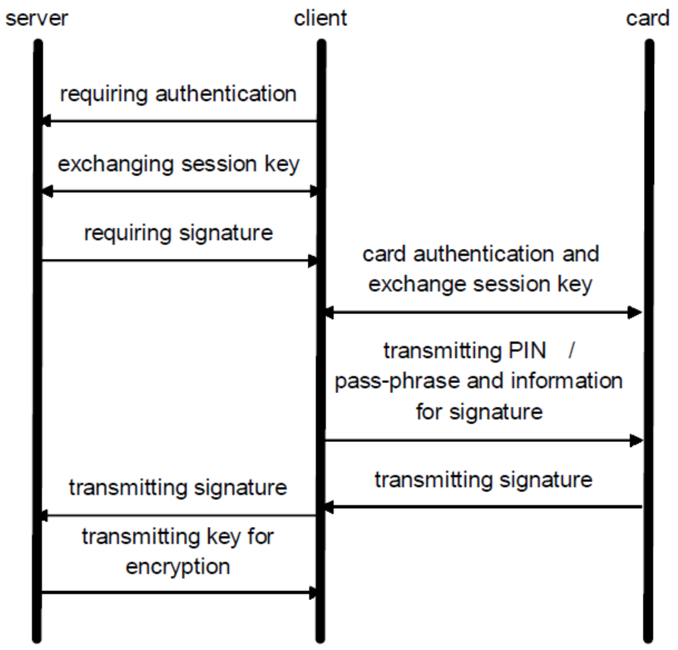

server
client
card
requiring authentication
exchanging session key
requiring signature
card authentication and exchange session key
transmitting PIN / pass-phrase and information for signature
transmitting signature
transmitting signature
transmitting key for encryption

Figure 3.4: Protocol - Extracted from [55]

In this system, the client must first authenticate himself in the server. After establishing a secure connection, the SD card proceeds to transmit it's signature to the server, to which the server replies with the encryption key. It utilises asymmetric encryption for the signature and symmetric encryption for the data files. This system also has an additional feature. If it detects an authentication brute force attempt, or physical tampering of the security areas, it will start to erase all the data within the file. This process cannot be stopped once it begins, even by cutting the power of the card because it will continue once power is reestablished. [55]

A secure solution for USB flash drives using the FAT file system, is another project that attempts to solve the same problem. This solution focuses on providing confidentiality, authentication, access control, efficiency and resistance against impersonation attacks. This system uses password based authentication, and a partition for the secure area and another for the normal area. The secure area is encrypted with a symmetric key. To authenticate the user a password is used, the system claims to not expose the location of where the password is stored. If the password is lost or forgotten it is possible to recover it with the help of a server that manages the passwords. [56]

The single chip solution of Embedded USB Encryptor [57] is an embedded solution to solve the problem of insecure data in USBs. The system uses a smartcard to store the encryption keys, and uses the AES algorithm to perform the encryption of data. The implementation of the AES algorithm is done at a physical level as can be seen in Figure 3.5, where there is a module dedicated to the Rijdael algorithm. It is also possible to see that the physical Encryptor chip stands between the computer and the USB device. The authentication process requires the smartcard to give access to a user. To sum the process, when a user wants to read or write to the USB, the smartcard must be presented and validated by the Encryptor chip. The chip then either encrypts or decrypts the data and sends it to its destination.

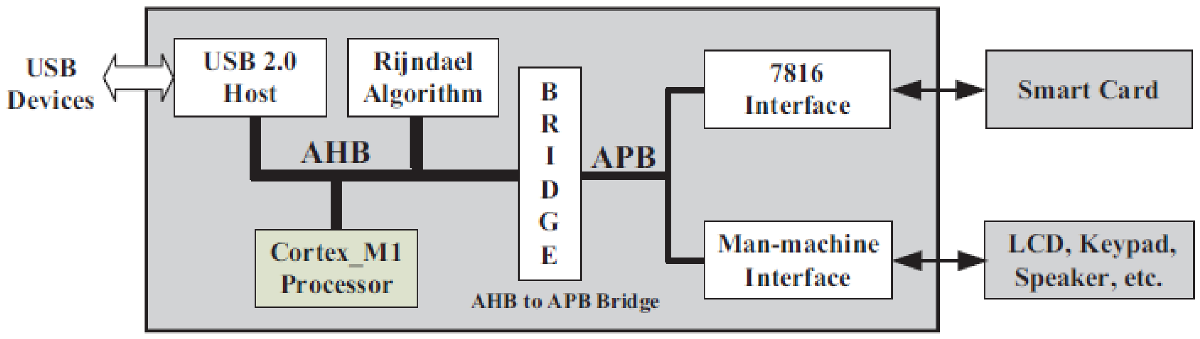


Figure 3.5: Structure of the Encryptor - Adapted from [57]

One solution to provide digital security to USB flash drives are the Secure USB drives. These USB drives offer authentication, confidentiality and authenticity. However in a study published in 2007 [58] these Secure flash drives often come with critical vulnerabilities that can be exploited to reveal sensitive information. The Secure USBs had an authentication system reliant on a password. A partition function to determine how much space would be for public access and which should be private. It also featured an initialisation function which was used to format the USB in case the user forgot the password. However, these methods of security were flawed. The communication between the USB and computer was not encrypted as such it was possible to discover the old and new passwords when inspecting the memory of the drive. The initialisation function does not delete data properly, it is possible to recover it after formatting the drive. The drives also have other vulnerabilities. This scenario is worrisome as users may think they have a secure method to protect their data, when in reality these solutions are quite vulnerable.

In 2011 another study of Secure USB devices [59] was published. This study showed that it was possible to sniff the packets being sent from the computer to the USB drive and capture the authentication information. It was also easy to reverse engineer the authentication procedure and discover the location of where the authentication information is stored in the memory. Some devices used the same password for authentication and encryption which made decrypting files possible once the password was discovered.

In ”The State of Embedded-Device Security” published in 2012, the article reports that millions of embedded devices connected to the internet have little to no security and can easily be hacked by malicious individuals. It also denotes that mobile devices are going to start to be targeted more often due to their increase in computational power. [60]

In 2016 a study that used Optical Fault Injection attacks against the flash memory of smart cards showed that it was possible to extract sensitive information from these devices. These attacks were carried out using a device capable of analysing the power consumption of the card, locating the flash region, identifying laser sensitive points and then performing the Bumping Attack [61]. By analysing the power consumption of the card it is possible to control the time to successfully execute attacks. The physical location to conduct these attacks is mapped by using optical fault injection. The final step is to confirm that is it possible to change the values of bytes when data is being transferred from flash memory to the registers. [62]

Today in 2020 the current solutions for this problem are encrypted USB drives, like the Kingston IronKey D300 or the Aegis Secure Key 3z. These USB drives offer data encryption with AES256-XTS and anti-tamper protection with the use data purging in the case of too many failed logins. However these drives offer no data integrity protection and also have very steep prices, even with small storage spaces like 4Gigabyte (GB). [63] [64]

4 Proposed Solution

Contents

In this section comparisons about different possibilities presented in the Background section will be analysed and the better candidate will be chosen for this project.

The objective of this project is to provide a low cost system that can guarantee digital security to flash based portable memory devices, like SD cards or USB flash drives. It is necessary to first develop an architecture of the system. the architecture should encompass three main criteria.

- Functionality;
- Digital Security;
- Ease of use.

Functionality is the most basic requirement for this project. It guarantees that the system is usable. The user must be able to issue commands and it must be possible to read or write to the SD card. This involves studying possible methods of transferring files, methods of communication between the microprocessor and the user. It is also necessary to study how the SD card manages file allocation and how the file system manages the files. The most basic functions that a user should be able to do are:

- Read;
- Write;
- Delete;
- List;

Other functions can also be, opening a file and editing it in real time, copying, cutting and pasting files. These functions are not crucial but also add functionality to the system.

Digital security is guaranteeing that the system protects the user data adequately. This involves studying which cryptographic algorithm is best suited for encryption. Studying ways of providing freshness, integrity and authenticity to the files. Also how the user should authenticate himself to the system.

Ease of use is how the user interacts with the system. It is a criteria dedicated to studying ways to increase the comfort of the user in interacting with the system. It has less priority then the other two but it can enrich the work done since the easier the system is to use, the more people can use it.

Figure 4.1 represents a diagram of the functions that are necessary to implement. The functions on the client are mere remote calls to the corresponding functions on the Arduino. They do not perform any critical actions. The functions on the user end only transmit or receive information, commands, or parameters like file size, filename, etc. The Arduino is responsible to execute all the critical actions such as encrypting files, writing them in an SD card or validating the user login. The SD card already has its built in functions, for this project the read, write and delete functions are the most important. The Arduino interacts with the SD card with the use of a file system. This file system is responsible for managing virtual block allocation and making API calls to the SD card.

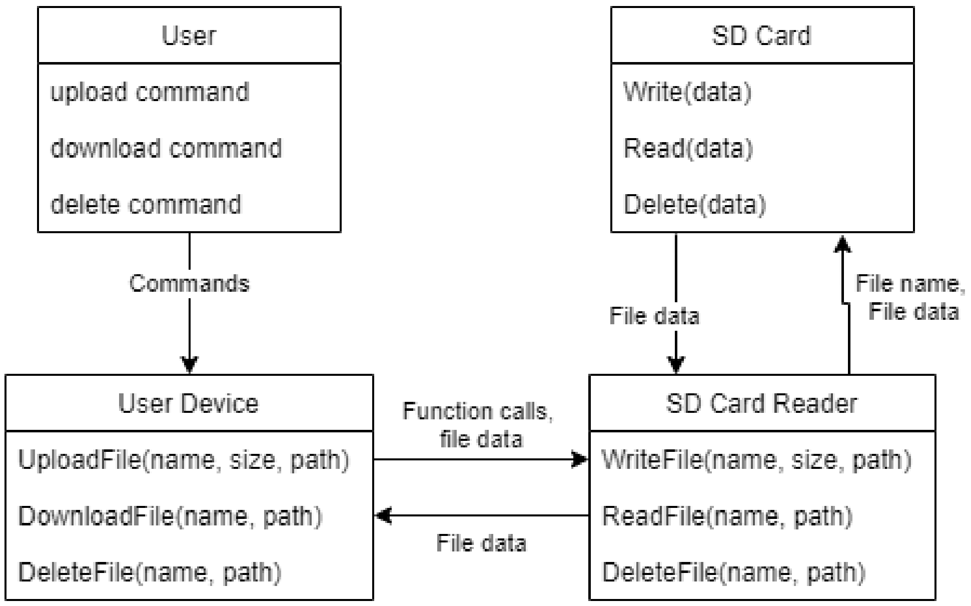


Figure 4.1: High Level Concept

4.1 High Level Concepts

As depicted in figure 4.1 the four main components of this solution can be observed. First the user which interacts with the system. The user only interacts with his user device (smartphone, laptop, etc..) by sending commands. this can be done by typing them in a terminal window, or by simply dragging and dropping files similarly to how a USB drive is used today. This depends on the level of development of the end-user interface.

The user device is the device that the user chooses to interface with the system. It does not do any processing related to the data. It only invokes functions on the SD card reader and sends or receives file data.

The SD card reader is the name used to designate the middle system between the SD card and the user device. The SD card reader, is the micro-controller that will do the bulk of the data management and processing. It is responsible for deciding how the files are written in the SD card by implementing a file system, encrypting the data before it gets written, validating and assuring the integrity and authenticity of the contents in the SD card. In short, it is responsible for reading, writing, deleting and assuring the digital security of the files in the SD card. In this solution we assume the SD card reader has limited resources, like small amount of RAM and low processing capabilities. As such, the transfer of the file will take significantly longer than transferring a file via USB.

The SD card is the portable storage device where the files get stored. It has built-in functions from its manufacturer. For this project the important functions are related to the allocation and writing of blocks. The way the SD card manages blocks is decided by the algorithms and heuristics the manufacturer has designed. It is beneficial to not interfere with these processes since the manufacturer of the card has expert knowledge of the physical composition of the card. Interfering with these processes can cause the blocks the become unevenly worn out or the write amplification factor might not be as efficiently controlled and this may lead to a large waste of space at the physical level in the SD card.

Figure 4.1 depicts the functions that can be executed by each member. These represent the basic functions necessary to guarantee that the system is usable. With these functions the user can utilise the SD card as a portable storage device, which is its intended purpose. Starting with the user, he is capable of sending commands to the interface. This interface can be a terminal window, where the user types commands or it can be a system window, where the user drags and drops files similarly to how a USB drive works. Regardless of the interface the user must be able to perform the read, write and deleting of files. Assuming the interface is on a bare-bones level, when a user wants to upload a file he needs to type the upload command and specify the file name and the path to be stored on the SD card. This procedure is similar to the download or deletion of a file. The command then gets sent to the program that is running on the user device. The user device needs to also collect the file size and then execute the UploadFile function with the parameters given by the user. Before starting the data transfer of the file, the user device calls the WriteFile function on the SD Card Reader with.

When the WriteFile function is called on the SD card reader, it starts by creating the file and making the necessary application programming interface (API) calls to the SD card to allocate the necessary number of blocks for the upload process. This is why send the file size is important, it allows for the pre-allocation of all the required blocks for the file transfer. Without this parameter the allocation would have to be dynamic with constant reallocation of blocks, this would mean a much slower transfer time. Because of this limitation, it is important to analyse different options of file transfer since the SD card reader will also have to encrypt and secure the file integrity.

It is necessary to define a proper protocol of communication for transmitting files. There are multiple ways to to this using for instance, User Datagram Protocol (UDP) based approach. It is also necessary to make it clear when the file transmission starts and ends. One way to accomplish this is by using an extremely uncommon character or combination of characters that would never be used in this scenario. However this method is not the most reliable since it may prove to be hard to find such an uncommon string. Another method would be to use the file size which is already being sent. This is a more reliable way, assuming there is no interference in the communication which could cause data loss. This method has the advantage of sending only the necessary information (since UDP does not care if the packets are well received), but has the downside of not being very reliable over communications that may have interference like wireless networks.

Instead of UDP based method we can use a Transmission Control Protocol (TCP) based approach. With this approach a syn-ack similar protocol is used. Each file is transmitted by being split into multiple small blocks, similarly to the previous method however, after each block is sent a confirmation is necessary. This method is much more reliable since it ensures that all blocks are received properly. It does have the disadvantage of slowing down the file transmission since each block needs to be confirmed. It is preferable to use UDP when dealing with a scenario of low CPU processing power or low bandwidth. [65] Since the processing power of the Arduino is limited, so the UDP based approach will be used.

A final option could be to simply transmit a constant stream of information from the computer to the SD card reader. This option is quite simple to implement but requires a large buffer of RAM for the embedded system to be able to hold the entire file.

For the encryption is is necessary to decide which algorithm is better suited to be used in a scenario of constrained resources. That will be discussed in a future section. All cryptographic file systems previously studied had to decide where to store the keys. Three alternatives were used. Either to store the key files on the SD card and encrypt all those keys with a master key. This has the advantage of being portable since to access all the files only a master key is required. The disadvantage is that this occupies useful space. Another possibility is to store the keys in the computer of the user. The advantage is that the SD card does not waste space with such files but the great downside is that the user must always access from his computer or store them in a separate device. The third option is to store the keys on a dedicated server for this purpose. The advantage is that it allows for portability and frees up space on the SD card. The downside is that this adds a significant material cost since it is necessary to pay for a server for this specific purpose.

For the integrity and authenticity there are some solutions. The file hash of the whole file is calculated outside of the microprocessor and then sent and encrypted, Each block that is sent to the microprocessor is authenticated, or a MAC is used to guarantee the integrity of the files. The first solution has the disadvantage of pushing code to the user-end, but is much faster on the transmission side. the second solution has the disadvantage of doubling the file size since each block of bytes has to be authenticated and validated by the embedded system which would slow down the whole process.

In terms of data freshness one solution that was studied in Chapter 3 was the approach used by the IRIS system. If we can adapt that solution to the resources available it is possible to assure data freshness. In this sense, the embedded system would have to hold a list that would store a user session hash, as well as the user storing it. Each time the user wants to login to the system these hashes would have to be compared to check if the system has been replaced by previous versions of itself, which render integrity and authenticity checks ineffective for this scenario.

This session hash can be calculated in different ways. The simpler method is to calculate the hash of the entire content of the SD card. This method is more expensive since everytime the user makes a change in a file, it is necessary to recalculate the hash of all the contents of the SD card. A way to reduce the processing time required for this hashing method if to create a single hash for each file and then hashing the concatenation of all the hashes. This way, when a file is altered only that singular hash needs to be recalculated and updated in the session hash. To be even more efficient, directories can also have their own hashes and the session hash is the concatenation of the directories. This structure is basically a tree of hashes, which allows for the validation and update of hashes in logarithmic time.

4.2 Interaction User-Interface

In this section the interaction between the user and the program and which type of interfaces are possible, authentication methods are also discussed.

The first option is via terminal. This is the simplest solution from the perspective of the programmer. The user would launch a program via terminal, and then select from predetermined commands which action he desires to perform. It is the solution that requires the least amount of coding by the programmer since it is just configuring a string based menu.

The second option is to add a level of abstraction to this implementation and create a visual window in Java, Python or another language that supports this type of graphics. This method is much easier for the user to utilise, he needs only to start the program and then it is just a matter of dragging and drooping files into the window.

A third option is to use a Dynamic Linking Library (DLL) in Windows, or Shared Object in Linux. This is the method that USBs, Keyboards, Mouses and other peripherals use to receive and transmit information to the computer. With a DLL the microprocessor can pretend to be a USB device. Once connected it would appear to the user as an external driver much like a USB drive or SD card would behave. This is the simplest option for the user since he wouldn't be required to perform any actions to communicate with the device. It would be plug and play. It has the disadvantage of being the most complicated in terms to programming or at least of similar complexity when compared to the second option.

Another possibility is to use the Arduino USBHost library, this library is specifically designed for the Arduino to appear as a USB device. It would be as simple as the third option for the user. However it has the great disadvantage of only being compatible with Arduino Due, or requiring a USB shield which would increase the financial cost of the solution.

To perform an authentication we can consider some possibilities. The first, is to use a login system dependent on a text password. Every time a user wishes to login he must insert his secret password. If a user wants to share the contents of the SD card with another person then all that is required is for the distribution of this secret password. Another method could be to use a physical token, with this method only those who possess a specific ID card, for instance, could access the contents of the files. It is also possible to combine these methods and form a dual factor authentication mechanism which is more robust. It is also possible to create an Access Control List and allow the SD card to be used by multiple users, with different passwords.

The user also needs to be able to navigate through the different directories that he creates. There are two ways to accomplish this, either the user writes the entire path every time he wishes to open a file or the Arduino keeps a track of which directory the user is in. The first option is much less user friendly as this means the user must constantly list directories and then write the whole path. The second option can be solved with a path variable. Due to memory constrains in the Arduino the path variable would have to be limited in it's size.

4.3 Communication PC-Microprocessor

In this section we will discuss different methods of communication that are possible between the Computer of the user and the microprocessor, this discussion will encompass both software and hardware.

One type of communication is via USB cable. This is a type of communication that is supported by some integrated boards, like Arduino, Raspberry Pi, Cubieboard and others. The USB communication is quite versatile since it is possible to find cables that convert USB A to USB B or C or micro USB, which means that the embedded system can even interface with an Android smartphone, for instance. It also has the advantage of allowing for reliable transmission of data, since there is no interference under normal conditions. It has the disadvantage of having a material cost, though not very costly, and also a distance limitation, since USB cables are generally short.

Another popular way of transmitting data could be via Wireless network. This solution is not as compatible as the USB cable since it sometimes requires the use of shields or other modules to support this type of communication. It is also typically slower than cable since it is more susceptible to interference. The big advantage is that the range of communication is large. WiFi communication can also be intercepted and is susceptible to man in the middle attacks [66].

On the software side there are multiple ways to transmit information. First the way commands and data are transmitted could be done in different ways. In a single channel implementation both commands and data are sent over the same socket. This also implicates that commands and data are sequential, a command will be sent, then data, then a command. This type of implementation is useful from single threaded applications. Since only one process is being worked at a time then it is pointless to create multi threads.

Another solution is to use a dual channel, or multi-channel where one socket is reserved for sending commands and others to send data. This implementation takes advantage of multi-threaded environments. For example, using a master-slave architecture with the master process receiving commands and creating slave processes as necessary. This would allow to read multiple files or write to different files, all at the same time. This approach is much faster than a sequential approach. To have real parallelism, at least a dual core processor is needed.

4.4 Hardware

For this project we will use an Arduino Uno. There are other more powerful variants like Arduino Mega or Due, but Uno has enough processing power and RAM to accomplish the requirements of this solution. It is also a cheap solution, which is important for this project since it aims to not make hardware a necessity or a deterrent. If the software developed can run on an Arduino Uno then it will be able to be scaled to any other type of hardware, like a smartphone, other more powerful micro-controllers, or even dedicated servers. By choosing such a restricted hardware environment we are guaranteeing that this solution is extremely scalable. This choice also does not restrict the user devices. The user has multiple choices of hardware to connect to the Arduino. It can be a desktop computer, a tablet, smartphone or any other thing capable of accepting USB connections. Another reason is also that Arduino features a large online community with help forums where many bugs and problems are discussed and resolved. This made the development of software easier, since many problems had already been addressed by others which reduced the time needed to resolve bugs. The online community is also quite active with constant bug fixing and software innovation. New libraries keep being created and added to the official forums. This helps to keep the solution maintainable and updatable in case it is necessary. The Arduino is connected to the user device via a USB cable, which also acts as a power cable for the Arduino.

The shield SD is the only shield capable of communicating with a SD card, so it was used to establish a connection between the SD card and the Arduino. The shield SD is connected via breadboard.

To simulate the user device a laptop was used. From this laptop the user interface is executed and files are transferred.

4.5 Data management by the SD card reader

The embedded system to be used has to manage the data that the user sends it. It is necessary to define how the encryption, integrity, authenticity, data freshness and even the writing of the files will be handled.

The communication to the SD card can be done in two ways, via a Shield or connecting the pins directly to the physical serial peripheral interface of the SD card. While feasible, the option of connecting the Arduino pins directly to the SPI involves opening the SD card and potentially rendering it useless. This procedure would also have to be done to any SD card that wishes to interface with the Arduino.

Using a shield it is possible to interface with endless cards without breaking them apart.

For writing on the SD card there are some possibilities. We can access the SD card in a raw manner, by simply writing to the blocks and sectors of the SD card. Another possibility is to create a library that implements a file system to decide how to best allocate the blocks. The third option is to use a library that is already created and satisfies the requirements of this project. In the case of utilising a library it is necessary to use a file system that is designed for flash memories. The file system should try to write to blocks in a uniform way, so that it does not cause premature failure of the blocks. It should also try to minimise fragmentation as this is a way to use more blocks than required and also causes a loss of performance.

4.6 File system comparison and choice

A good file system for this work should meet a set of requirements. Have a light implementation, due to Arduino's memory constraints the file-system's library should not use a lot of RAM memory in its functions. Have almost non-existing file fragmentation, fragmentation causes more memory blocks to be allocated than necessary, this may reduce the SD card lifespan. Good performance on write, read and delete functions, the file system should not be a bottleneck for the whole system. Encryption is a plus, if the file system already comes with security functions like encryption, it is a desirable trait. In Table 4.1 the main advantages and disadvantages are described.

	Pros	Cons
Ext4	Low fragmentation, Good performance	Low compatibility
NTFS	Good performance	High fragmentation, limited compatibility
JFFS2	Good file compression	Slow mount times, extremely low compatibility, garbage collection reduces flash memory lifespan
YAFFS	Fast mount times	Extremely low compatibility, old file system
F2FS	40% more efficient than Ext4, low file fragmentation	Extremely low compatibility, unstable due to insufficient testing and large updates
FAT32	Highest compatibility	4 GB maximum filesize, filenames limited to 8 characters
exFAT	Best performance, unlimited filesize, long filenames	Less popular than FAT32, larger implementation than FAT32

Table 4.1: File systems Pros and Cons

Even though NTFS and Ext4 are good file systems and widely used by Windows and Linux, they are not well suited for flash devices due to incompatibility problems, ext4 is only used for Linux operating systems. NTFS has the added problem of file fragmentation which should be avoided in flash based devices to maximise their durability. F2FS is a new system, that shows some promise however it still has some instability issues and so it does not seem like a reliable choice for this project. JFFS2 is an efficient system that uses compression to better utilise the memory available, but it suffers from slow mount times. Also the fact that SD cards are getting bigger in terms of memory with SD cards now reaching 128 GB, compression of files is not exactly a priority. YAFFS only has one advantage, the fact that it has quick mount and unmount times, but in this project this has not been a problem with the already tested FAT32 so not a very compelling reason to adopt this file system as the preferred alternative. FAT32 is a good candidate since it has a relative good performance, but it is a file system that is getting old and showing its limitations, like only allowing 4GB as the maximum file size. exFAT is the best choice. It has an improved performance when compared with FAT32 and fixes many limitations of the previous system. It also has good compatibility with most operating systems.

4.7 Encryption Algorithm comparison and choice

Triple DES is an ill suited candidate since it has a lot of vulnerabilities and is considered a weak cipher. The MARS algorithm also has a complex software implementation and vulnerabilities when implemented in smartcards which is makes it impossible to be considered for this project. The other algorithms all have poor performances when compared against Rijndael.

Taking into consideration the work analysed in Subsections 2.2.1 and 2.2.2 the best algorithm to perform encryption algorithms taking into account the requirements of this project is AES. The algorithm has the best performance overall. It is the most resilient against attacks, being capable of resisting even known plain-text attacks. On top of being chosen has the world's standard for encryption by NIST, it is also the fastest algorithm, having outperformed all other algorithms in terms of not only speed, but also having used the less amount of RAM, and having a small implementation size, in a smartcard. This test is very relevant, since a smartcard has very limited resources, which is similar to the environment of this work, since an Arduino Uno has also very limited resources. This makes it the indisputable choice for this project.

4.8 Cipher mode comparison and choice

ECB is not an adequate choice for this project since the microprocessor needs to encrypt files, which mean large amounts of blocks and that would require large amounts of keys, for a secure level of encryption. CFB, OFB and CTR are stream ciphers, they have the disadvantage of not being able to guarantee integrity and also when a change is made to the data they require complete re-encryption, because of this are not a good choice for this project.

While promising, OCB is shown to have vulnerabilities in respect to the authentication phase, so it will not be considered for this project.

CBC is an algorithm that is vulnerable to tampering. Even without knowing the key, the attacker can switch the blocks of the cipher-text so that when it gets decrypted, it will still make sense. It's also not possible to use multi-threading with CBC, but that is not an issue in this project, since microprocessors use only a single core. When a change is made to the data CBC requires only that the data after the new modification has to be re-encrypted, the previous data does not need to change, this is a clear advantage over stream ciphers.

One solution presented to solve this problem is to use a diffuser before the encryption takes place. First we use a hash algorithm that hashes the plain-text and then we encrypt the resulting hash. This supposedly makes CBC less prone to tampering. This is the solution employed by Microsoft with the Elephant diffuser [67]. However there is still the question of just how secure the hashing algorithm is. In 2015 Microsoft removed the Elephant diffuser from it's full disk encryption procedure and eventually switched to XTS.

XTS has some disadvantages, the fact that is needs two keys instead of one, means that extra space has to be used. XTS is also usually used for full hard-drive encryption, but can also be used in secure encrypted flash drives. XTS is also vulnerable to tampering which means that it is also possible to alter cipher-text blocks ciphered with XTS and it will still make sense when decrypted, this problem can only be resolved with the use of integrity checks. It can be fully parallelized but a microprocessor cannot really benefit from this aspect. It allows for random access writes to be made. This is an important feature since flash memory tends to degrade faster than hard-drives and so it's useful to minimise the amount of needless re-writes to maximise the longevity of the SD card. The way XTS ciphers blocks is also an advantage over CBC. With CBC every block is chained, which means that if a block is corrupted or tampered everything past that block will be decrypred incorrectly. With XTS if a block is corrupted or tampered, only that block will be decrypted incorrectly, since the blocks are chained. The disadvantages of are easily outweighed by its advantages, especially since keys are quite small, using only 128 bits, considering that SD cards keep getting larger in size, this disadvantage has a tendency to become irrelevant. It is approved by NIST, and has interesting and useful features that set it apart from the others. As such XTS was chosen for this project since it was the best cipher from all the studied ones.

4.9 Hashing Algorithm comparison and choice

MD5 is a fast and extremely popular algorithm, but its use is not advised since it is clearly vulnerable, as such it can not be considered for this project since it would compromise integrity. Tiger is an algorithm that also has vulnerabilities, it will not be considered for this project either since it cannot also guarantee integrity. SHA256 and SHA512 are good candidates for this project, they offer good security and performance, and are considered the standard for hashing functions at present time. Whirlpool is an efficient algorithm in constrained scenarios as such it is a good candidate for this project.

The choice of use for this project is to use SHA256 for guaranteeing the integrity of the files. For full file integrity checks using SHA256 will suffice as it is still considered the industry standard.

4.10 Solution Proposal

Taking into consideration the state of the art discussed and compared, and the related work done we can now propose a solution for the problem at hand.

The file system used by the SD card will be exFAT. Arduino will use AES with the XTS cipher mode to encrypt file contents and BLAKE2 for file metadata encryption. To guarantee file integrity and authentication SHA256 will be used to create HMACs of the files to be encrypted. To encrypt each file a random key will be used, this random key must be created after an upload and be as random and unpredictable as possible. There is no need to store the keys, only the pass-phrase used to create them. These pass-phrases must me mapped to their respective files in a file tree which will be encrypted with the user's master key. As implied, the SD card will be used to store the tree file. The master key is created from the user's password. The password is also the way in which the user authenticates the files he or she uploads. To guarantee freshness Arduino will also keep a tree file stored in its internal memory where a user session hash is stored, this hash can also be created using BLAKE2. This allows Arduino to manage multiple users since its internal memory can hold up to 1 KB of data, and each entry only takes up a few bytes.

4.10.1 Overview

Figure 4.2 describes how the system would process the upload of a file to the SD card. First the user writes in the terminal, that is executing the program to communicate with the Arduino, the desired command. His device, in this case a laptop, relays that command to the Arduino. The process of transferring data begins, the Arduino receives chunks of data from the laptop and encrypts those chunks before writing them into the SD card. This process loops itself until the entire file is transferred. Then the Arduino sends an ACK to acknowledge that the file has been received without problems. The user can then select another command, from the ones showed to him in the interface of the terminal.

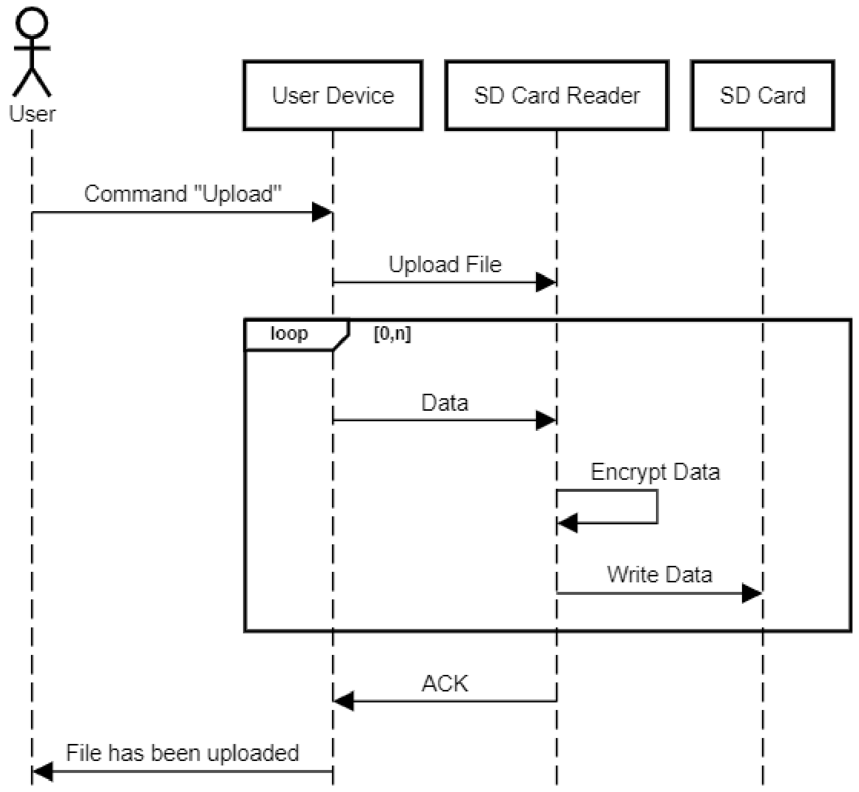


Figure 4.2: Overview Diagram

4.10.2 Detailed view

Figure 4.4 we can see a more detailed view of how the file upload takes place. First the user must login to the system. After his identity is verified by the Arduino board, the user can issue commands. In this case the user decides to upload a file by writing ”up” in the terminal console. The program that communicates with the Arduino asks the user what is the filename of the file. After the user specifies the filename, both the filename and file size are sent to the Arduino. A file with the same name is created in the SD card. The laptop starts sending chunks of 16 bytes of the file to the Arduino, which encrypts those chunks and writes them in the SD card. This process loops until the entire file is transferred. The hash is computed and written alongside the file. The confirmation is sent back, that the file has been uploaded successfully. The user then wishes to logout. This is transmitted to the Arduino which proceeds to compute the session hash and encrypt it and write it in its internal memory as well as in the SD card. The Arduino logs out the user and is ready to be used by another person or with another SD card.

In Figure 4.3 it is possible to see the tree structure used to guarantee integrity. Inside the folder SYSTEM, a replica of the file system is created. But only file MACs and folder MACs are stored. The R_MAC is the root’s MAC which is copied to the internal memory of the Arduino to guarantee data

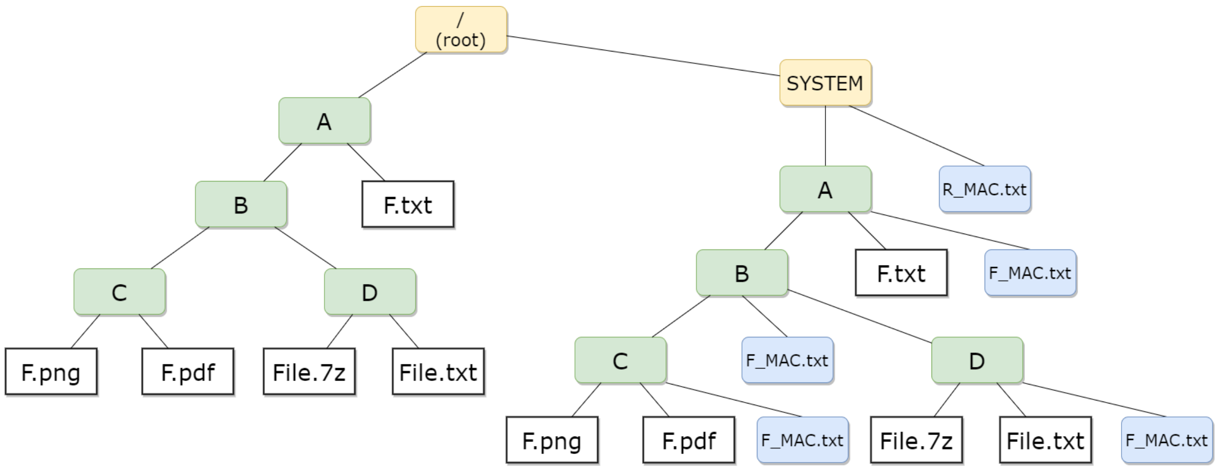


Figure 4.3: Integrity tree structure

freshness. This replica of the file system allows the reading, insertion and deletion of file MACs to be done in O(1). It is only necessary to add the prefix ”SYSTEM/” to the path defined by the user and the file MAC is obtained. The calculation of the folder MAC is done in O(n).

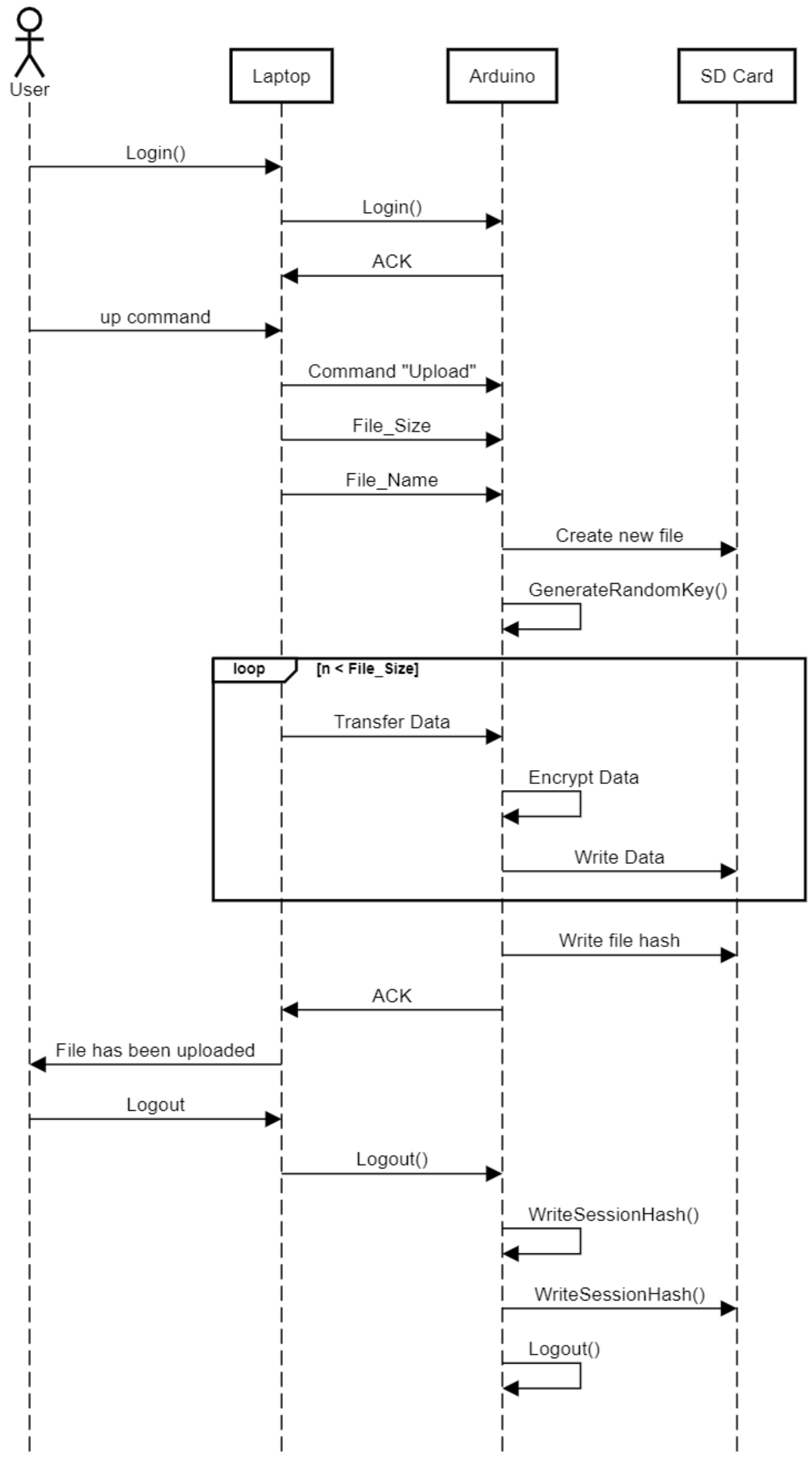


Figure 4.4: Secure File System diagram

4.11 Solution Comparison

This subsection is for presenting the proposed solution based on the pros and cons discussed previously. The Best Solution describes the best solution that is possible to achieve given the restrictions of this project. The minimal solution describes the minimum requirements that must be met in order to ensure usability, functionality and security. These solutions serve as baselines for the prototype which will fall in between these two solutions.

4.11.1 Best Solution

The system is plug and play. Figure 4.4 represents the functions necessary to implement for the ideal solution. The user connects the Arduino via USB to his user device and would use it as a regular USB device. In this interaction he would be able to upload, download, delete and edit files. This would be achieved with a DLL file. The DLL file would have the code necessary for allowing the Arduino to pass as a USB device. In this solution the user simply drags and drops files as if he was using a normal USB drive. The user device simply performs data transfer operations. It also sends the file path, filename and file size to the Arduino. The Arduino then starts encrypting the file and writing it to the SD card. For the encryption the AES256 algorithm with XTS cipher mode will be used. Each file will be encrypted with a different and ransom key, the keys are all encrypted with a master key from the user, which is given to the Arduino at the beginning of a session. After the writing process is conclude, it is necessary to guarantee the integrity, authenticity and freshness of the file. This is done by creating a MAC of the file which is a concatenation of the file hash, author's name and file version. The file hash is also a concatenation of multiple hashes. These hashes are calculated from different sections of the file. By dividing the file in sections and creating a hash of each section it allows for a much faster recalculation of a new hash in case there are changes to the file. For calculating file hashes SHA512 will be used. The author's name is the current user's name, and the file version is a counter that is incremented each time there is a change in the file. The hashes will be calculated after file encryption, this makes the reading process much more efficient since it is not necessary to decrypt the file in order to validate the file integrity, authenticity or version. Each directory will hash a MAC which is a concatenation of all the MAC's of its files including other directories. This allows for a much faster reading. When a user wants to read a file, the validation will be done following a tree-like structure. When a user wants to delete a file the file blocks will be completely overwritten with random information, including the file's key. This ensures the file information is completely destroyed. When a session ends, the root MAC is stored in the internal memory of the Arduino.

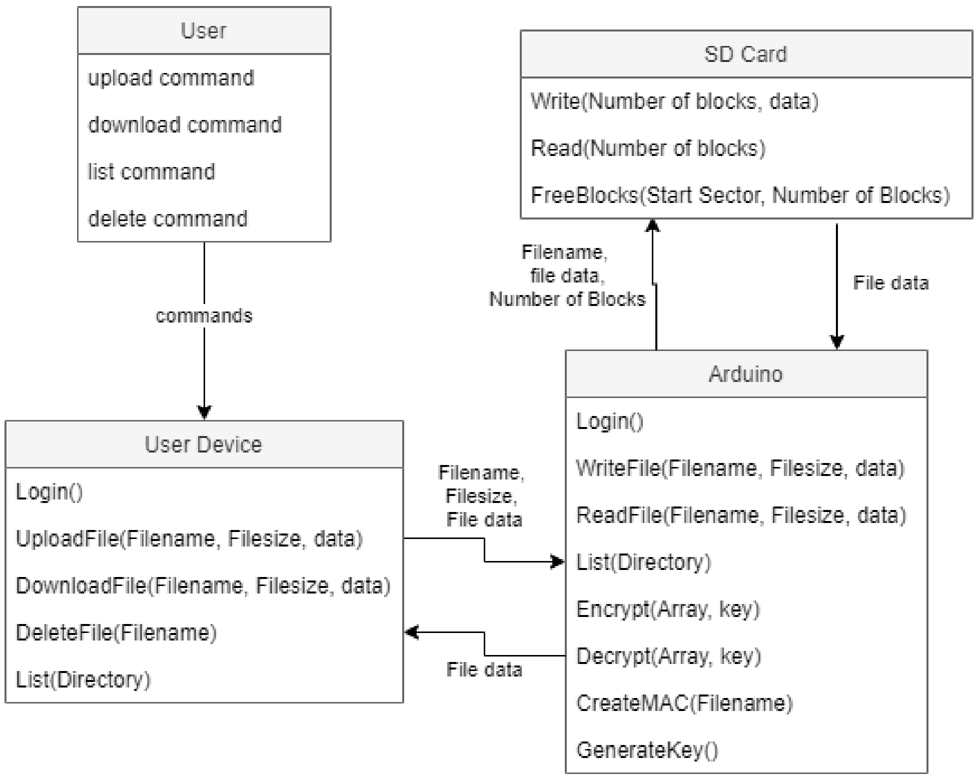

User
upload command
download command
list command
delete command
commands
SD Card
Write(Number of blocks, data)
Read(Number of blocks)
FreeBlocks(Start Sector, Number of Blocks)
Filename, file data, Number of Blocks
File data
Arduino
Login()
WriteFile(Filename, Filesize, data)
ReadFile(Filename, Filesize, data)
List(Directory)
Encrypt(Array, key)
Decrypt(Array, key)
CreateMAC(Filename)
GenerateKey()
User Device
Login()
UploadFile(Filename, Filesize, data)
DownloadFile(Filename, Filesize, data)
DeleteFile(Filename)
List(Directory)
Filename, Filesize, File data
File data

Figure 4.5: Pseudo UML functions diagram

4.11.2 Minimal Solution

The user will have to be able to send commands to the Arduino, this can be achieved with a terminal interface where these commands are typed. The basic commands are upload, download and delete. When a user wishes to upload a file to the SD card he types the command into the terminal. The user device then invokes the upload function from the Arduino providing it with the file path, filename and file size. It then starts sending the file data in chunks. The Arduino starts receiving and encrypting the data. For encryption AES128 with CBC cipher mode will be used. Each file has to be encrypted with a random key and all the file keys encrypted with the master key. After the file data transmission ends the file hash is calculated by using SHA256. The file hash is then concatenated with the user's name and the result is encrypted with the file key creating the file MAC. The root MAC is the concatenation of all the file MACs, the root MAC is stored in the Arduino internal memory. When a user wants to delete a file, only the file key is rewritten.

4.12 Development

In this section the development methodologies used are presented. This work follows a three iteration process, where each iteration adds significant changes and functionality to the solution.

4.12.1 First Iteration

The tasks of the first iteration consisted in, studying and choosing the most appropriate file system library to use, creating file transfer functions specifically upload, download and file listing. Multiple file systems were analysed during this task. The first library used was the SdFat library. This library is compatible with exFAT but it is still in beta. While it was possible to create and write files with it, it had some bugs. It also had a heavy implementation, which when combined with the security libraries, would use too much RAM and the Arduino would simply not work. The next libraries used were FatFS and PetitFS. FatFS was quickly dismissed since it provided no advantage over the default SD library. The PetitFS library has a very small implementation size since it only included basic functions, however it only supported FAT32 or FAT16, it also only capable of writing files of up to 512 bytes. Since there was no library which supported exFAT and was lightweight, the default SD FAT3 library was used. It is the most tested and stable library so it is a good candidate.

In the first iteration of the solution it was necessary to create everything from scratch. Even though Arduino has a large online community with lots of guides, there was no work done on transferring entire files to the SD card via Arduino. The only similar work done consisted in sending keystrokes to the Arduino via its own built-in terminal. It was necessary to develop two scripts. One on the Arduino side which will become the main program. The second script is client side and is used to receive and transmit user commands and file data. During this iteration the main priority is to be able to transfer files between the SD card and the Arduino, encryption is the second priority.

The language used for the Arduino is C++, there is no option. In the client-side script Python 3 was used, mainly because of the easiness in managing sockets and files. The client side script interacts with the user by presenting a list of commands the user can choose and also when to write the filename and path. It uses the serial library to send and receive file data. For the Arduino script the native FAT32 and Serial libraries were used in this phase.

When developing the algorithm to send the file it was necessary to send small blocks of the file to the Arduino. It was also necessary to calibrate the rate at which these blocks were sent since the Arduino processes information at a much slower rate than a desktop CPU. The file size was also used to determine when the transaction was terminated. Initially a character was used, but this method proved unreliable since the file transmission would sometimes end prematurely. It was difficult to find a special character or sequence of characters to determine the end of file transmission. The file size method proved much more reliable with no margin for error. After it was possible to transmit and receive files.

4.12.2 Second Iteration

The second iteration focuses on providing the necessary digital security requirements. This is the iteration where the encryption, file integrity, authenticity, data freshness, secure deletion will also be added. The tasks for this iteration involve creating the necessary functions to guarantee the security requirements. The implementation of the security algorithms was achieved with the use of a library. The library used was Arduino Cryptography Library. This library provides an extensive number of not only encryption algorithms but also integrity functions to chose from. As such it is a vastly superior choice when compared to the other options available which frequently only offer a single algorithm. The library is public, this increases its effectiveness since the community can help to verify and correct potential bugs in the implementation. For this reason it is preferable to use a public implementation of AES-XTS rather than creating it from scratch.

After selecting the library the encryption was added. The encryption and decryption calls are called in between the reception and the writing of a file chunk. When a file chunk is received by the Arduino, the block is encrypted and only then written to the SD card. With the encryption working, the next step was to add integrity. The integrity is done by creating a hash and updating it with the encrypted file chunk. After the entirety of the file is received, a file MAC is computed. The MAC consists of a concatenation of the hash of the encrypted file and its file key, both encrypted with the user's master key. However this is when difficulties related to RAM space started to occur. The Arduino Uno's RAM was being used past its limit and the program was not executing properly. The creator of the library suggested to delete some functions that were not being used and needlessly occupied RAM space. After performing the necessary changes to the library, there was enough RAM space to create and validate file MACs alongside the encryption. However it was still not possible to create and validate folder MACs. This is because to do so, it is necessary to have two open files, the directory and the file MAC, and also to calculate the hash. This problem persisted through weeks which forced the Third iteration to be started before the conclusion of this task. Eventually it was possible to complete this task by disabling compiler optimisation for specific functions. This allowed for less RAM to be used in trade for execution speed. The secure deletion implementation consists of rewriting the file MAC with its decrypted contents generated with a random key, the file is simply removed. To guarantee data freshness, the root MAC is copied into the Arduino's internal memory in every log off, and is verified in each log in.

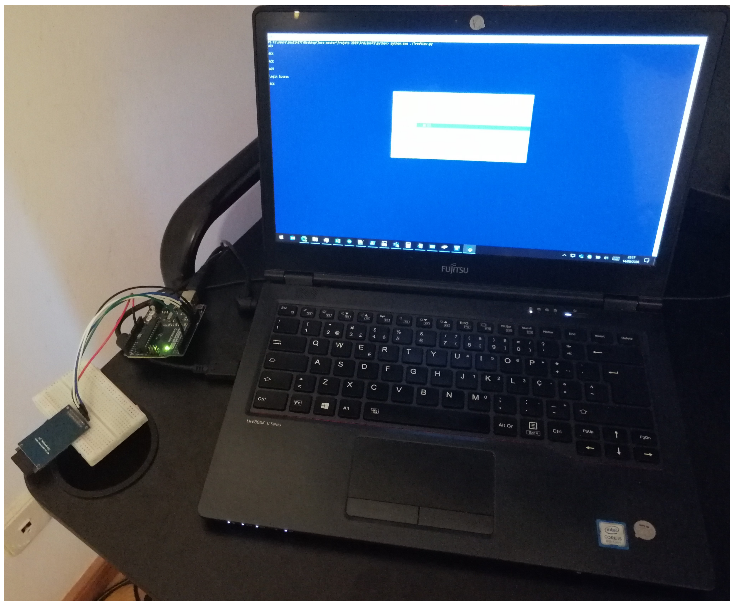
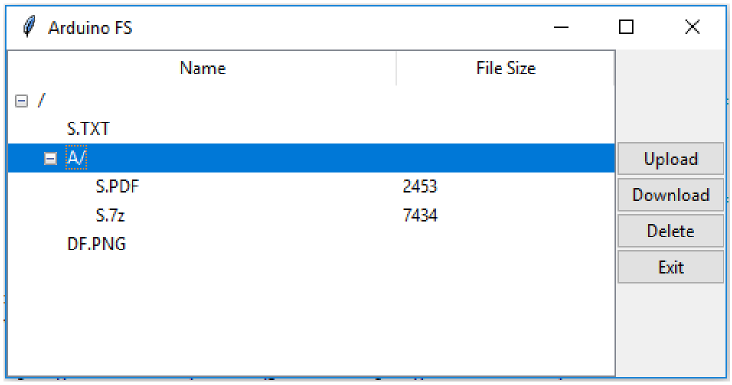


Figure 4.6: Full system

4.12.3 Third Iteration

The third and final iteration consisted of bug fixing and the creation of a simple Graphical User Interface (GUI). This GUI allows the user to see the contents of its SD card in a tree like structure, which is updated after every file transaction. The GUI, which can be observed in Figures 4.7 and 4.8, also allows the user to press buttons to send commands. The main advantages of using the GUI are the fact that the user no longer needs to write the commands in a terminal, which reduces input errors. Makes users more comfortable when using the system since the information stored in the card is update in real time. The users no longer have to rely on prints to verify the contents of the SD card.



Name	File Size
/	
S.TXT	
A/	
S.PDF	2453
S.7z	7434
DF.PNG	

Figure 4.7: Graphical User Interface

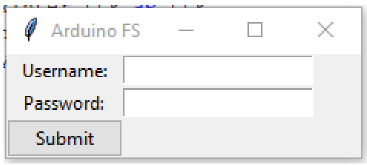


Figure 4.8: Login Menu

5 Results and Discussion

Contents

In this chapter the testing methodology used is described and explained. The results are showed analysed and discussed.

5.1 Test Methodology

The tests conducted focused on the transfer times and transfer rates. These measure the overall performance of the system. It is also possible to analyse the impact that security has on simple file transfer. Three scenarios were used. Using the full system, which features file transfer, encryption and the integrity tree structure described in Subsection 4.10.2. Using only the encryption functions. Using no security at all, simply transferring files. Additionally, the times were measured transferring files in both ways, from the laptop to the SD card, and from the SD card to the Arduino. Nine different file sizes were used, starting from 500 bytes and increasing exponentially up to and including 128.000. bytes. This allowed to see how the system behaved and allowed to measure rigorous times that were not affected by potential hardware variations, or security overheads. Each file size was measured one hundred times for each scenario and transfer direction. The results presented in Figures 5.1 5.2, are the averages of each test. For each test the file transfer time was measured. With the transfer times it is possible to calculate the transfer speeds measured in bytes per second. When transferred to the SD card each file was transferred to an empty directory. This means that the folder verification and updating was at its possible lowest time when using the full system. Since it is also necessary to measure the evolution of how long it takes to verify the integrity of folders, new tests were conducted. These focused on transferring a file to a folder in the SD card which contained ninety nine files. After each transfer the file would be deleted. Inversely the times were also measured of transferring a file from a folder with one hundred files in the SD card. Each of these tests was also conducted one hundred times. A script was created to simulate the inputs, the GUI was not used for these tests.

5.2 Result Analysis

In the case of the Arduino Uno, its maximum baud rate is 115200 units, via Serial. This means that the Arduino Uno is capable of sending and receiving a total of 115200 bits per second which equates to 14400 bytes per second. This value is not an achievable number since it doesn't take into consideration parity, redundancy, error bits or other mechanisms used by the Serial protocol. Rather it is a value which we want to get as close to as possible.

In the laptop to SD card transfer rates, the security operations overhead is noticeable. By encrypting the information as it is received the transfer rate is reduced by approximately 30%. With encryption and integrity checks and computations, we see that the transfer rate suffers a reduction of approximately

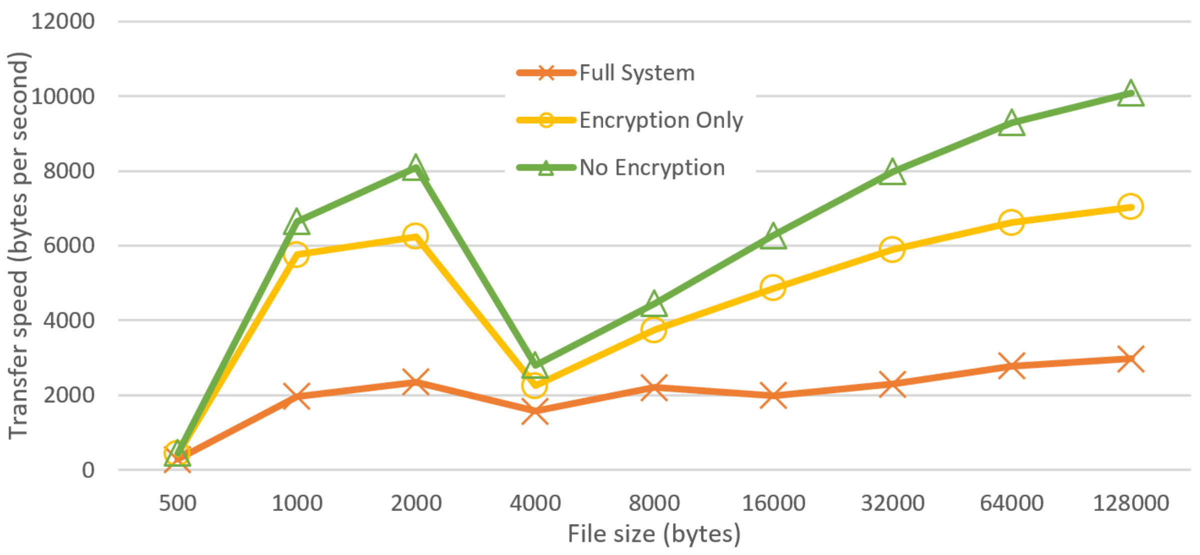


Figure 5.1: Transfer Rates (Laptop to SD card)

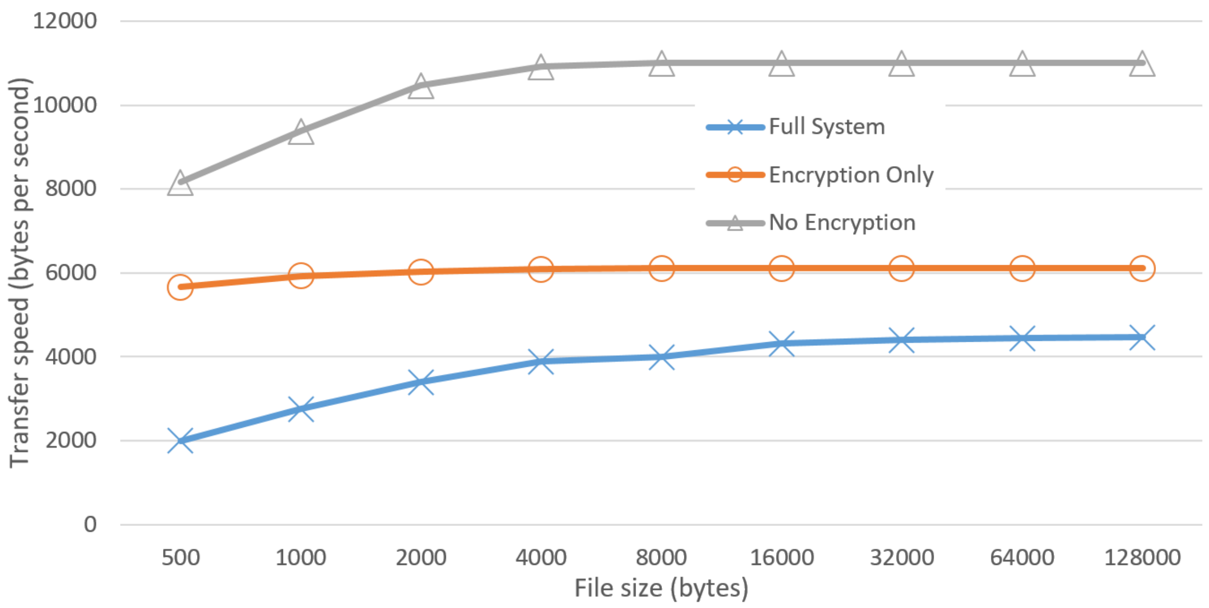


Figure 5.2: Transfer Rates (SD card to Laptop)

70%. This penalty is due not only to the fact that the data is being encrypted and the hash is being computed in real time, but also because it was necessary for some functions to turn off compiler optimisations to save RAM space in exchange for execution time. The slow file transfer times with smaller files can be attributed to file operations. The transfer time was measured until the interface received the final confirmation from the Arduino that the file has been closed and properly written to the SD card.

The transfer to the laptop are more stable than the opposite rates relatively to smaller files. This is explained by the fact that the computer used for this test has a much more powerful processor, than that of the Arduino. This facilitates file management operations which causes the overheard to be less noticeable. The download rates where only encryption was used and encryption plus integrity are similar due to the functions being the same. The only difference are the file and directory verification operations which are done before the transfer cycle initiates. This overhead gets less noticeable as the file size grows.

Overall, the maximum file upload rate is approximately 10091 bytes per second, while the maximum download file transfer rate is approximately 10997 bytes. This means that these file transfer operations achieve approximately 70% and 76% of the baud rate maximum transfer capacity. As expected the more security operations are performed the bigger the overhead which will translate into slower transfer times. The discrepancy between upload and download transfer rates is due to the fact when uploading files the Arduino Uno has to send a confirmation to the computer so that the next file chunk is sent. During the download the Arduino continually sends file chunks since the laptop can receive information at a much faster rate than the Arduino can send it.

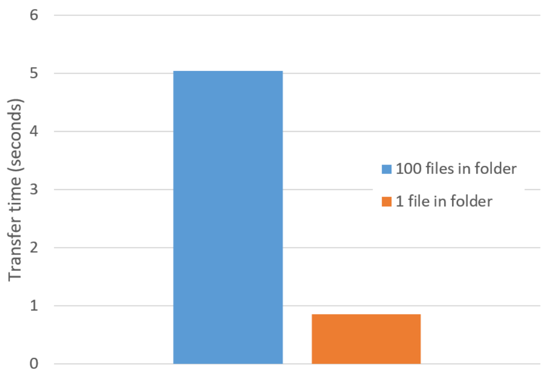


Figure 5.3: Folder MAC verification and update comparison

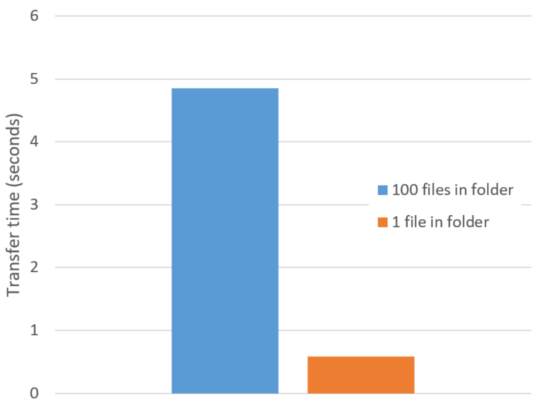


Figure 5.4: Folder MAC verification comparison

The tests conducted in Figure 5.3 consisted in sending a file to a folder with one hundred files in the SD card. Figure 5.4 represents the inverse test of transferring from the SD card to the laptop a file from a folder that contains one hundred files. Both tests were each conducted one hundred times like the previous ones, with a file of 2000 bytes which allows us to calculate the time it takes for the

Arduino to calculate the folder MAC with n files. The formula to calculate a folder MAC and update it is $time = 0.0419 * n$. The formula for calculating the folder MAC is $time = 0.0426 * n$. With these times it is possible to calculate the time it takes to transfer a file of any size, to or from a folder with n files. The formula for transferring a file from the laptop to the SD card is

$$time = 0.0419n + 925.79ln(x) + 723.99$$

The formula for the time of the inverse operation is

$$time = 0.426n + 1188.7ln(x) + 2048$$

Where n is the number of files in the targeted folder, and x represents the file size in bytes.

The growth of time increases logarithmically. This happens since there are discrepancies that occur at multiple levels such as hardware fluctuations, communication socket delays, or when writing files to the SD card. When dealing with small file sizes and extremely low transfer times, any disturbance will affect the transfer speed greatly. All these influence the execution of the code, however, when the file sizes increases the transfer time of those disturbances becomes less significant and the growth curve starts to stabilise. This growth is asymptotic and will eventually stabilise in a value near the Arduino's maximum transfer. This equation shows that for sufficiently large files the software and algorithms developed are not a bottleneck. The only way to increase the transfer speed is to use more powerful hardware. The transfer speeds of writing to the Arduino are also more unstable since the laptop only transfers more data to the Arduino after receiving the signal and writing being more intensive than reading. Since writing involves altering the memory of the SD card. Meanwhile, when transferring data to the laptop, the Arduino does not wait for any confirmation and can continuously send data.

The statistics of the software used are 30196 used for flash memory (which stores the code to be executed and constant variables), this value represents 93% of all available flash memory that the Arduino Uno uses. The system starts its execution with a total of 1113 bytes of RAM occupied (54% of all RAM available). RAM was the main bottleneck is this work and forced the redesign of algorithms, usage of different libraries or other alternatives. This shows that the system is being used at full capacity and as mentioned previously the only bottleneck of this project is the Arduino.

6

Conclusion

Contents

In the conclusion section we will talk about the overview of this work as well as the results obtained.

6.1 Conclusions

The objective of this work was to create a file system that provided digital security to flash based portable devices, using a micro-controller to handle the data management and the security operations. The micro-controller chosen was an Arduino. It is a fairly limited hardware. Despite its constraints, it was possible to implement a solution that used a file system that took into consideration the necessities of flash memory while also achieving a robust secure system. The system is able to encrypt the data with a secure encryption algorithm and cipher while also guaranteeing the integrity of each file, including directories. By doing so, the system also guarantees the freshness of the data, being impervious to attacks that use previous versions of files.

In relation to the system performance, the full system takes a heavy hit in performance, due to having to use last resort measures, such as disabling compiler optimisations. The system developed has practical applications in the real world, capable of being used in a personal or corporate environment.

6.2 System Limitations and Future Work

The system has some limitations, such as the user not being able to create files named F_MAC. Users also can't edit the files in the SD card. To edit or update a file it has to be transferred from the SD card to the user device and edited there. The Arduino Uno flash memory is full, it is not possible or at least advisable to continue to add code since it may lead to code malfunction.

In terms of future work there are multiple pathways. To convert the system to a library to be used publicly by the community. To refactor the code and optimise it. To continue to add more security operations. To scale the system to a more powerful system, such as an Arduino Mega and add more features. Another improvement to the project is to also implement real time file editing.

Bibliography

[1] https://pixabay.com/vectors/laptop-black-blue-screen-monitor-33521/, accessed: 2019-09-11.

[2] https://pixabay.com/vectors/microchip-computer-electronics-23313/, accessed: 2019-09-11.

[3] https://pixabay.com/vectors/memory-card-sd-secure-digital-98414/, accessed: 2019-09-11.

[4] Mikben, "File system functionality comparison - win32 apps." [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/fileio/filesystem-functionality-comparison

[5] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2, 2007, pp. 21–33.

[6] W. A. Bhat and S. Quadri, "Review of fat data structure of fat32 file system," *Oriental Journal of Computer Science & Technology*, vol. 3, no. 1, 2010.

[7] D. Woodhouse, "Jffs: The journalling flash file system," in *Ottawa linux symposium*, vol. 2001, 2001.

[8] C. Manning, "How yaffs works," *Retrieved April*, vol. 6, p. 2011, 2010.

[9] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2fs: A new file system for flash storage," in *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, 2015, pp. 273–286.

[10] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009, p. 10.

[11] E. Barker, W. Barker, W. Burr, W. Polk, M. Smid *et al.*, *Recommendation for key management: Part 1: General*. National Institute of Standards and Technology, Technology Administration, 2006.

[12] J. Kelsey, B. Schneier, and D. Wagner, "Key-schedule cryptanalysis of idea, g-des, gost, safer, and triple-des," in *Annual International Cryptology Conference*. Springer, 1996, pp. 237–251.

[13] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management part 1: General (revision 3)," *NIST special publication*, vol. 800, no. 57, pp. 1–147, 2012.

[14] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[15] R. J. Anderson, E. Biham, and L. R. Knudsen, "The case for serpent." in *AES Candidate Conference*, 2000, pp. 349–354.

[16] R. L. Rivest, M. J. Robshaw, R. Sidney, and Y. L. Yin, "The rc6tm block cipher," in *First Advanced Encryption Standard (AES) Conference*, 1998, p. 16.

[17] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas, L. O'Connor, Mohammad, Peyravian, D. Safford, and N. Zunic, "The mars encryption algorithm," 1999.

[18] F. Sano, M. Koike, S.-i. Kawamura, and M. Shiba, "Performance evaluation of aes finalists on the high-end smart card." in *AES Candidate Conference*, 2000, pp. 82–93.

[19] M. Dworkin, "Recommendation for block cipher modes of operation. methods and techniques," National Inst of Standards and Technology Gaithersburg MD Computer security Div, Tech. Rep., 2001.

[20] T. Krovetz and P. Rogaway, "The ocb authenticated-encryption algorithm," Tech. Rep., 2014.

[21] N. Ferguson, "Collision attacks on ocb," *Preprint, February*, 2002.

[22] L. Martin, "Xts: A mode of aes for encrypting hard disks," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 68–69, 2010.

[23] "Aes-xts block cipher mode is used in kingston's best secure usb flash drives." [Online]. Available: https://www.kingston.com/en/solutions/data-security/xts-encryption

[24] J. Kim, A. Biryukov, B. Preneel, and S. Hong, "On the security of hmac and nmac based on haval, md4, md5, sha-0 and sha-1," in *International Conference on Security and Cryptography for Networks*. Springer, 2006, pp. 242–256.

[25] F. Chabaud and A. Joux, "Differential collisions in sha-0," in *Annual International Cryptology Conference*. Springer, 1998, pp. 56–71.

[26] R. L. Rivest, "The MD5 Message-Digest Algorithm," Network Working Group Request for Comments (RFC) 1321, Apr. 1992.

[27] X. Wang and H. Yu, "How to break md5 and other hash functions," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2005, pp. 19–35.

[28] R. Rivest, B. Agre, D. Bailey, C. Crutchfield, Y. Dodis, K. Elliott, F. Khan, J. Krishnamurthy, Y. Lin, L. Reyzin, E. Shen, J. Sukha, D. Sutherland, E. Tromer, and Y. Yin, "The md6 hash function a proposal to nist for sha-3," 11 2008.

[29] W. Penard and T. van Werkhoven, "On the secure hash algorithm family," *Cryptography in Context*, pp. 1–18, 2008.

[30] J. Kelsey and S. Lucks, "Collisions and near-collisions for reduced-round tiger," in *International Workshop on Fast Software Encryption*. Springer, 2006, pp. 111–125.

[31] P. Barreto, V. Rijmen *et al.*, "The whirlpool hashing function," in *First open NESSIE Workshop, Leuven, Belgium*, vol. 13. Citeseer, 2000, p. 14.

[32] S. Corporation, *SanDisk Secure Digital Card Product Manual*, 2003.

[33] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano, "The design and implementation of a transparent cryptographic file system for unix." in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 10–3.

[34] R. Geambasu, J. P. John, S. D. Gribble, T. Kohno, and H. M. Levy, "Keypad: An auditing file system for theft-prone devices," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 1–16.

[35] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," in *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5. ACM, 1999, pp. 124–139.

[36] K. K. Mar, Z. Hu, C. Y. Law, and M. Wang, "Secure cloud distributed file system," in *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*, 2016, pp. 176–181.

[37] S. Liang and B. S. Rawal, "Secure usb based file system for bmc applications," in *2017 Second International Conference on Recent Trends and Challenges in Computational Models (ICRTCCM)*, 2017, pp. 228–233.

[38] H. Ji, J. Feng, M. Liu, and X. Yang, "Design of usb storage encryption device based on xts-aes," in *2017 9th International Conference on Intelligent Human-Machine Systems and Cybernetics (IHMSC)*, vol. 2, 2017, pp. 137–140.

[39] J. Bonwick and B. Moore, "Zfs: The last word in file systems," 2007.

[40] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: A zfs case study." in *FAST*, 2010, pp. 29–42.

[41] A. Oprea and M. K. Reiter, "Integrity checking in cryptographic file systems with constant trusted storage." in *USENIX Security Symposium*. Boston, MA;, 2007, pp. 183–198.

[42] A. Oprea, M. K. Reiter, and K. Yang, "Space-efficient block storage integrity," in *In Proc. of NDSS '05*, 2005.

[43] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, "Iris: A scalable cloud file system with efficient integrity checks," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 229–238.

[44] F. Apolinário, M. Pardal, and M. Correia, "S-audit: Efficient data integrity verification for cloud storage," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018, pp. 465–474.

[45] M. D. Corner and B. D. Noble, "Zero-interaction authentication," in *Proceedings of the 8th annual international conference on Mobile computing and networking*. ACM, 2002, pp. 1–11.

[46] M. Kaminsky, G. Savvides, D. Mazieres, and M. F. Kaashoek, "Decentralized user authentication in a global file system," in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 60–73.

[47] Y. Li, L. Du, G. Zhao, and J. Guo, "A lightweight identity-based authentication protocol," in *2013 IEEE International Conference on Signal Processing, Communication and Computing (ICSPCC 2013)*, 2013, pp. 1–4.

[48] T. Shah and S. Venkatesan, "Authentication of iot device and iot server using secure vaults," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2018, pp. 819–824.

[49] N. Meenakshi, M. Monish, K. J. Dikshit, and S. Bharath, "Arduino based smart fingerprint authentication system," in *2019 1st International Conference on Innovations in Information and Communication Technology (ICIICT)*, 2019, pp. 1–7.

[50] R. Perlman, "File system design with assured delete," in *Third IEEE International Security in Storage Workshop (SISW'05)*. IEEE, 2005, pp. 6–pp.

[51] N. Joukov and E. Zadok, "Adding secure deletion to your favorite file system," in *Third IEEE International Security in Storage Workshop (SISW'05)*. IEEE, 2005, pp. 8–pp.

[52] B. Lee, K. Son, D. Won, and S. Kim, "Secure data deletion for usb flash memory." *J. Inf. Sci. Eng.*, vol. 27, no. 3, pp. 933–952, 2011.

[53] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, “Reliably erasing data from flash-based solid state drives.” in *FAST*, vol. 11, 2011, pp. 8–8.

[54] P. Zhang, W. Zhang, S. Niu, and Z. Huang, “User level secure deletion for usb flash disks,” in *The 2014 2nd International Conference on Systems and Informatics (ICSAI 2014)*, 2014, pp. 1072–1077.

[55] T. Kato, T. Tsunehiro, M. Tsunoda, and J. Miyake, “A secure flash card solution for remote access for mobile workforce,” *IEEE Transactions on Consumer Electronics*, vol. 49, no. 3, pp. 561–566, 2003.

[56] S. Lee, K. Yim, and I. Lee, “A secure solution for usb flash drives using fat file system structure,” in *2010 13th International Conference on Network-Based Information Systems*, 2010, pp. 487–492.

[57] Hanlin Chen, “The single-chip solution of embedded usb encryptor,” in *2010 IEEE International Conference on Information Theory and Information Security*, 2010, pp. 42–45.

[58] Hanjae Jeong, Younsung Choi, Woongryel Jeon, Fei Yang, Yunho Lee, Seungjoo Kim, and Dongho Won, “Vulnerability analysis of secure usb flash drives,” in *2007 IEEE International Workshop on Memory Technology, Design and Testing*, 2007, pp. 61–64.

[59] K. Lee, S. Pho, and K. Yim, “Reversability assessment on secure usb memories,” in *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2011, pp. 6–8.

[60] J. Viega and H. Thompson, “The state of embedded-device security (spoiler alert: It’s bad),” *IEEE Security Privacy*, vol. 10, no. 5, pp. 68–70, 2012.

[61] S. Skorobogatov, “Flash memory ’bumping’ attacks,” vol. 6225, 08 2010, pp. 158–172.

[62] F. Cai, G. Bai, H. Liu, and X. Hu, “Optical fault injection attacks for flash memory of smartcards,” in *2016 6th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, 2016, pp. 46–50.

[63] “Ironkey d300 secure usb 3.0 flash drive - 4gb–128gb - kingston technology.” [Online]. Available: https://www.kingston.com/en/usb-flash-drives/ironkey-d300-encrypted-usb-flash-drive

[64] “Aegis secure key 3z.” [Online]. Available: https://apricorn.com/aegis-secure-key-3z

[65] A. Milanovic, S. Srbljic, and V. Sruk, “Performance of udp and tcp communication on personal computers,” in *2000 10th Mediterranean Electrotechnical Conference. Information Technology and Electrotechnology for the Mediterranean Countries. Proceedings. MeleCon 2000 (Cat. No.00CH37099)*, vol. 1, 2000, pp. 286–289 vol.1.

[66] D. P. Andito, D. F. W. Yap, K. C. Lim, W. K. Yeo, and T. H. Oh, “Wireless sensor networks coverage: An error performance analysis,” in *The 17th Asia Pacific Conference on Communications*, 2011, pp. 836–839.

[67] N. Ferguson, “Aes-cbc+ elephant diffuser: A disk encryption algorithm for windows vista,” 2006.