A Password Capability-Based System with an Integrated Capability-Based Cryptographic File System

ACRYLICS

By
Jørn Guldberg

Master's Thesis
Department of Mathematics and Computer Science
University of Southern Denmark
June 2021

Supervisor: Prof. Daniel Merkle
Co-Supervisor: Prof. Joan Boyar

SDU

DEPARTMENT OF MATHEMATICS
AND COMPUTER SCIENCE

PRINT: GRAFISK CENTER, SDU

Dedication

ACRYLICS

Advanced Cryptographic (Li) Capability-Based System

Where Li stands for Lisbeth. My mother. Who is no longer with us.

ACRYLICS and the thesis are dedicated to her,
and her name is placed in the middle of the ACRYLICS name.

She was a positive and hardworking person to the very last days of her life. We lost her one and a half years before I started at the University. If she had not passed on her super powers to me, I wouldn't have had the powers to hand in a Master Thesis, maybe not even to start at the University.

I'm very proud of my work.

Til dig Mor

Acknowledgements

Prof. Daniel Merkle, SDU IMADA
I would like to thank Daniel Merkle for being my supervisor. You have been a key person throughout my time at the university. Both in courses I attended, to individual study activities, an industrial project and as your teaching assistant. Your amount of support has been a huge driving force, and I'm very grateful for that. I'm thrilled that you agreed on my thesis topic and your guidance and feedback through the thesis has been priceless.

Prof. Joan Boyar, SDU IMADA
I would like to thank Joan Boyar for being my co-supervisor. I have also attended several of your courses, including an individual study activity. These were some of the most challenging courses during my degree and lot of hard effort was done. Especially in the cryptology course. I already knew back then that I hopefully should use the knowledge in my thesis. I was therefore thrilled that you agreed to be my co-supervisor. Our discussions and your feedback through the thesis has also been priceless.

Benjamin Larsen, PhD Student, DTU Compute
I would also like to thank Benjamin Larsen for our discussions on Trusted Computing and TPM's. Benjamin invited me to DTU for discussion of several things related to my thesis. It was very nice to be welcomed and valuable to have a discussion partner with a lot of experience and knowledge in the area.

Family and colleagues
I would also like to thank my family for a lot of support, and the possibility and room to have great ambitions. My teaching assistant colleagues and fellow students, for our discussions and for your support. I really like the ambitious environment we have created. My colleagues at my student workplace at Dubex, for discussions and support.

Resume

IT- og cybersikkerhed bliver i stigende grad en større og større bekymring. Det er estimeret at i 2020 der vil den årlige omkostning for cyberforbrydelser overstige 1 billion dollars totalt pr år. Imidlertid har de nuværende operative styresystemer der bruges til at beskytte data, brugere og tjenester, deres oprindelse og kernedesign fra styresystemer fra 1980'erne. Trods at cybersikkerhed som koncept først blev introduceret i 1999, forventes det stadig at de nuværende styresystemer er sikre. Nuværende styresystemer gør primært brug af "access control lists" til at håndtere sikkerheden, selvom at det er kendt at den "capability"-baseret tilgang kan tilbyde en større sikkerhed. Nuværende systemer har også "super-bruger designet", hvor en super bruger altid har tilladelse til alting. Der stilles derfor spørgsmålstegn ved sikkerheden af nuværende styresystemer og ACRYLICS foreslås som et nyt kernedesign for styresystemer. ACRYLICS er et komplet "capability"-baseret system og har et integreret kryptografisk filsystem, som også forsikre at den fysiske gemte data er gemt sikkert, selv når systemet er slukket. ACRYLICS' sikkerhed er underforstået af at følge opsætninger fra topmoderne forskningsresultater som er bevist sikre. Gennem definitionen af fire sikkerhed niveauer, bliver ACRYLICS vist til at være mere sikker end nuværende styresystemer og det forklares hvorfor at det hævdes at det er de nuværende styresystemers kernedesign der gør dem sårbare. En reference implementation af ACRYLICS er konstrueret og designet af ACRYLICS er vist til at virke i praksis.

Abstract

IT- and cyber security are continuously becoming a greater concern. It is estimated that in 2020 losses due to cybercrimes exceeded a total of 1 trillion dollars pr. year. However, the current operating systems used to protect data, users and services, have their roots and core design from operating systems designed in the 1980's. Despite Cyber Security as a concept only being introduced in 1999, we still expect the current operating systems to be secure. Current operating systems primarily rely on access control lists for security, even though it is known that a capability-based approach offers more security. Current systems also have the "super-user design", where a super user always has permission to everything. The security of current systems core design is questioned and ACRYLICS is proposed as a new core design for operating systems. ACRYLICS is a completely capability-based system, and has an integrated cryptographic file system, which also ensures the capabilities are stored safely offline. ACRYLICS' security is implied by it following schemes from state-of-the-art research results that are proven secure. Through the definition of four security levels, ACRYLICS is shown to be more secure than current systems and why it is believed that the current systems core design makes them vulnerable. A reference implementation of ACRYLICS is constructed, and the ACRYLICS design is shown to work in practice.

Contents

1 Introduction

IT- and cyber security are continuously becoming a greater concern [9]. Almost all systems are getting connected to the internet and it is a difficult task to protect data, users and services [80, 14]. It is estimated that in 2020 losses due to cybercrimes exceeded a total of 1 trillion dollars pr. year [24, 54].

Often it is only cyber security which is mentioned, but IT-security should not be forgotten. To obtain cyber security, IT-Security is required [78]. If we think about it, how should we obtain cyber security if the IT-systems we are building on are not secure?

A lot of improvement for cyber security has been seen in recent years, but nothing has really changed in the operating systems core design since the 1980's and it is necessary to take a closer look into the security mechanism in the current operating systems core design.

The current and most known operating systems have their origin from the late 80's and 90's and are often built on existing kernels which are even older. For example the Windows NT kernel was originally introduced in 1989 and in 1997 Microsoft announced their 64-bit version of the NT kernel [69]. It is the kernel still used by Microsoft and Windows 10 today. It has not been clear what the relationship between Windows NT kernel and Microsoft's earlier MS-DOS kernel (1981) is. We might have to see them as separate kernels. however, MS-DOS was build on QDOS (1980), where there is indications that QDOS was a copy of CP/M (1974) [83, 84] and there are still design decisions made back in the MS-DOS (or QDOS / CP/M) which are still present in Windows 10 today. If one tries on a Windows machine to create a file with the name ”con.txt” this will be denied due decisions traced back to the earlier kernels [82] which is the point here. Linux was announced in 1991 and the first version was released in 1994 [36], and it is still the same code base that is being developed today. Linux is built from Minix (1987) and Unix (1970) [12, 72, 71].

It is important to understand the timeline of operating systems and cyber security. The first publication about cyber security came in 1999 [9, 18], this is later than the initial development of most of the current systems, but we still expect them to be secure. The security criteria today is much more sophisticated than it was back then [44, 74].

While developers of course tries to update the current operating system to fit the needs, this is very challenging because of the complexity, legacy, technical debt and design choice made very early on [27, 55, 61, 3].

All of the largest known operating systems are primarily using access control list as access control, even though it is known that the capability approach offers greater security [26, 51]. The early capability systems Hydra [41] and CAP [53] was some of the earliest capabilities-based systems and there has been ongoing research in capability since, but Hydra and CAP were never widespread and there has never been a widespread capability based operating system.

Another security concern for the existing system is the core design where there exists privileged super-users and non privileged users i.e the systems have a root or admin user which can access anything and have all permissions to do everything. Such a user often overrules any security feature and undermines the complete security of the system.

This raises the questions, if the core design used for operating systems today makes them vulnerable? and is it possible to design an operating system core which can obtain better IT-Security and also be proven secure?

To answer the questions the most known operating systems and their state-of-the-art security mechanisms are analyzed, including other security mechanisms such as disk encryption tools, cryptographic file systems and secure hardware modules.

The Security Level model is introduced, and based on the analysis of the current systems Security Level 1 and Security Level 2 are defined. ACRYLICS' design will be proposed as a change in the core design for operating systems, and is reaching the better Security Level 3. ACRYLICS is based on state-of-the-art research results which proves security in core functionality of the system. With additional improvements ACRYLICS will reach the best Security Level 4.

Certain attacks will be discussed and are also the basis for the definition of the security levels. A reference implementation of ACRYLICS is presented and examples of the running system are shown, which validates the theory and design works in practice.

1.1 Contribution

In this thesis the security of the core design in the current and most known operating systems is questioned. By analysis and comparing to the Security Level model, see Figure 1.1, it is shown why the existing systems are vulnerable to certain types of attacks.

With the theory of Lopriore [42] and Damgaard and Dupont [21] combined, ACRYLICS is proposed with an improved core design for operating systems. ACRYLICS is a password capability-based system with integrated capability-based cryptographic file system. By following the schemes in [42] and [21], which are proven to be secure, ACRYLICS' security is implied.

The combination of the aforementioned schemes makes it possible to store the capabilities safely encrypted on the disk. Furthermore the capabilities are needed to read and decrypt the disk, i.e. the disk cannot be mounted on any system without both a boot-password and user credentials.

To the best of our knowledge no system has the same core design as ACRYLICS, and where the capabilities are stored securely physically on a disk when the system is powered off. Furthermore no system has been observed where encryption truly happens per user basis, and where shared files are encrypted with keys per file basis, derived from a user which has an active capability. All happens as default as a part of the core design.

By comparing ACRYLICS in the Security Level model, it is seen that ACRYLICS defend against the attacks defined in Security Level 2 and achieve the better Security Level 3. With a couple of improvements, ACRYLICS with improved capability-module reaches Security Level 4.

A reference implementation has been constructed and shows how the design works in practice.

To quantify the security of ACRYLICS compared to others, the Security Level model is used. It consists of four security level as follows:

Security Level	Description	Restricts super-users	Section
Level 4: ACRYLICS with improved Capability-Module	Only vulnerable for attacks where a valid user and with active capabilities is malicious. Only files shared with the malicious user can be compromised. Because of the capability-based security, there is no real super-user, and the system prevents privilege escalation where a single super-user can access anything.	✓	5.7.2
Level 3: ACRYLICS	Safe against outsiders but prone to a malicious valid user. The malicious valid user can compromise all shared files in the system with, even files not shared to the malicious valid user itself. Because of the capability-based security, there is no real super-user, and all user's private files are safe.	✓	5.7.1
Level 2: Full Disk Encryption	Safe against physical outsiders as long the encryption key is safe. Is prone to escalation attack due to the super-user design and the fact that encryption is happening transparent and only with one master encryption key.	✗	4.4.2
Level 1: No Disk Encryption	Vulnerable to multiple attacks. As the disk is not encrypted the data can always be read directly from the physical disk. Is prone to escalation attack due to the super-user design. The system's disk can be mounted to another system where the attacker is super-user, hence the attacker is super user and can access anything.	✗	4.4.1

Figure 1.1: Security Level model. Used to compare of security levels by system type.

1.2 Structure

In Chapter 2 general access control is introduced with focus on the difference between access control lists and capabilities. Lopriore's [42] extended password capabilities are introduced and are later used as a core module in ACRYLICS.

In Chapter 3 the security notation for disk encryption and Damgaard and Dupont's [21] disk encryption schemes are introduced. The disk encryption scheme is later used for the core design of ACRYLICS.

In Chapter 4 the current and most known operating systems will be investigated, analyzed and discussed. In particular their state-of-the-art security features will be analyzed. Also other security tools such as disk encryption tools, cryptographic file systems and the secure hardware module ”TPM” are analysed and discussed. At the end of the chapter security levels 1 and 2 are defined according to the Security Level model seen in Figure 1.1. The analysed systems are compared to the security levels and examples of different known attacks for each security level are also given.

In Chapter 5 ACRYLICS is designed with the knowledge gained from theory introduced in chapters 2 and 3, and the analysis in Chapter 4. Later in Chapter 5 two improvements to the design are proposed, leading to ACRYLICS with improved Capability-Module. At the end of the chapter Security Level 3 and Security Level 4 are defined. ACRYLICS' design is discussed and compared to these security levels.

In Chapter 6 ACRYLICS' reference implementation is introduced and in Chapter 7 we will see and discuss examples of the running ACRYLICS reference implementation.

2 Access Control and Capabilities

In this chapter the principle of least privileged and the two main classical access control models for operating systems; Access Controls Lists and Capabilities [66] are introduced. We will later use capabilities as the foundation of the access management control in ACRYLICS. In particular we will use the e-capability scheme described in Lopriore [42]. In Section 2.3 we do a recap of the e-capability scheme.

2.1 General Access Control

The main goal for operating system protection is the principle of least privileged [60]. It is a design principle where only the necessary permissions are given to someone to complete a task. One of the biggest security problems is when someone or something has more privileges than needed. Exploits do often take advantage of these ”extra” permissions, which leads to privilege escalation, where someone obtains more privileges than was meant to. To prevent privilege escalation a fine grained access control mechanism is needed, such that only the least amount privileges needed to perform the task can be granted, hence, using principle of least privilege.

2.2 Privileges: Access Controls List vs. Capabilities

In operating systems access control models are often divided into two main groups; Access Control Lists (ACL) and Capabilities.

ACL is the far most used approach and is used in the largest known systems like Windows, Linux and macOS [66, 59]. The advantage with ACL’s is that they are easy to handle. If we consider a Linux system, the UNIX permissions bits for a file is the access control list, because it describes who and what access the owner, group and others has to the file, this makes it easy to see who has which permission. These bits can also easily be modified to give, revoke or restrict access rights.

Within ACL and access control there are two different policies that can be used; Discretionary Access Control (DAC) and Mandatory Access Control (MAC). In DAC the owner of an object is the one, which can change and control who has access to an object. Classically, Windows and macOS used DAC. However, the trend is going toward using MAC, since it offers better security and more control. MAC are static policies which specify access permissions. So only if policies permits a user, the user can access a file or can change the access rights. Often MAC makes use of ”labels”, for example the labels ”non-secret”, ”secret” and ”top-secret”, then MAC can specify that a user with the ”secret” label can access ”secret” files and ”non-secret”-files but not the ”top-secret” files. The default in Windows is DAC but has a MAC layer on top, which checks permission before the DAC layer. Likewise in macOS, MAC is used invisible to the user to manage access control behind the scene and only in user space, the user sees the classical DAC. In most Linux distributions it is pure DAC, but for example with the SELinux patch, a MAC layer can be enabled.

The other approach is the Capability model [42, 17]. Where subjects (sometimes referred to as domains, as in [66]) have some capability to some object. The objects could for example be files, resources or actions. Subjects can for example be users, programs or services. A capability is like a key for a locked object, and only with the correct capability the object can be "unlocked" and accessed with the given permission. This means that it is possible to control what action or part of the object that are allowed to be accessed with a given capability, this gives a very fine grained access control, and is what is needed to achieve the Principle of least privilege.

One major difference between ACL and capability is where the permissions are stored design-wise. Either with the object or the subject. In [66] an example of an access matrix is given where subjects (domains, in the reference) are represented as the rows and the objects as the column. In ACL the permission is stored with the object, hence represented with the columns, whereas capabilities are stored with the subjects, hence represented with the rows. Another example is also nicely expressed in [26].

There are especially three myths about capabilities and capability based systems which many believe are true.

- ACL systems and capability systems can provide equivalent security
- Capability systems cannot enforce confinement
- Revocation of capabilities cannot be achieved

However, all these myth can be demolished [51] and for example with the confused deputy problem [31], where a user via a program with admin rights can run unintended commands as admin, is a classic example of the limitation of ACL and why capability should be used. The problem has been known since 1988 but there still does not exist a widespread capability based system today, even though the two research systems from the early 1970's Hydra [41] and CAP [53] existed at that time. Hydra and CAP were some of the earliest systems using capabilities, but were never widespread and since then ACL systems have taken over, almost completely.

2.3 Extended Password Capabilities

We have now discussed different access management schemes and will now dig into the e-capability (extended password capabilities) scheme from [42] which is later used in the design of the capability module in ACRYLICS. All information in the rest of this section is from [42].

Classically a capability C is defined as a pair of (B, AR), where B is the identifier of an object and AR is the set of access rights the capability has to that object. Objects have an object type T, where T defines the access rights for that type. Examples of object types could be files, resources or actions. Classical capabilities were improved by password capabilities. A password capability P is a pair of (B, W), where B is the identifier of a Capability-Object and W is a password. A different password should exist for any given access right, so a different W can lead to different access rights. However, the classical password capability approach can lead to password proliferation, and we need another improvement to be smarter about this. This is where the extended passwords capabilities (e-capability) comes into play.

2.3.1 e-capability

An e-capability E is a triple (B, W, R), where B is an object identifier, W is a password and R is the reduction field.

The reduction field specifies which access rights the capabilities have and is validated with W. To specify access rights the reduction field is divided into sub-fields, with a bit for every access right for the specific object type. To follow the example from [42], we assume we have an object of type "File" which has four access rights; EXECUTE, READ, WRITE and DELETE. The reduction fields have n-1 sub-fields where n is the number of access rights, and each sub-field has a size of n bits.

So a possible reduction field could look like this, where r_i is the sub-fields and e, r, w and d are the EXECUTE, READ, WRITE and DELETE access rights respectively.

```
   r2        r1        r0
[1,1,1,1]-[1,0,0,1]-[1,1,1,0]
 e r w d   e r w d   e r w d
```

Figure 2.1: Reduction field example

We calculate the access right as the result of a logical AND operation on the sub-fields:

$$AR = r_2 \wedge r_1 \wedge r_0 \tag{2.1}$$

Furthermore it should be noted that a sub-field containing only ones is referred to as a flat sub-field, e.g. r_2 is a flat sub-field in the above example.

In the following subsections we will go into detail about how W is derived from the object's password, how to validate if R and W grants the wanted access right on object B, how a e-capability can be reduced and lastly how revocation mechanisms work.

2.3.2 Password Derivation

We will now see how W is derived. First we should know how an object is created. As mentioned, objects are of a given object type T and T defines the AR for the object type. Furthermore the object has an owner password, which is denoted W_{own} and is a random generated bit string.

When a new capability object is created, a special owner capability is constructed, denoted E_{own}. E_{own} is given to the subject which is about to create the new capability object.

The special E_{own} capability consists of all sub-fields being flat, hence the capability has all AR, which also makes intuitive sense since it is the owner. If we continue the "File" example from before, the reduction fields of E_{own} will look like this.

```
   r2        r1        r0
[1,1,1,1]-[1,1,1,1]-[1,1,1,1]
 e r w d   e r w d   e r w d
```

Figure 2.2: Reduction field example for owner capability

Beside the reduction field, E_{own} also has a password. The password is the same password as the owner password in the correspondent object. This makes sense if we look at the password-derivation function, see pseudo code example in Listing 2.1.

The derivation function is an iterative process and it will either stop when reaching the first flat sub-field or have iterated over all sub-fields. In the case where we have E_{own} where all sub-fields are flat, the process will immediately stop and the return value will simply be the object's owner password.

Let's now consider the case where a reduced capability has to derive the password W. Let's use the reduction field from Figure 2.1. The process will start from the capability object's owner password (e.g. `out_password` = object's owner password) and enters the loop. Since the sub-field in question is not flat, the value of the sub-field is written into the encryption buffer, the buffer is encrypted with the current value in `out_password` and the value in `out_password` is overwritten with the result of the encryption.

In our particular case the next sub-field is flat, hence we break out the loop and return the current value in `out_password`.

```
derive_password(reduction_filed, capability_object)
{
    out_password = capability_object->password;
    for (int i = 0; i < reduction_filed->number_reduction_fields - 1; ++i)
    {
        // Check if the subfiled is flat
        if (flat_subfield(reduction_filed[i]))
        {
            break;
        }

        encryption_buffer = reduction_filed[i];

        // Current out_password used as key to encrypt encryption_buffer
        out_password = encrypt(encryption_buffer, out_password);
    }
    return out_password;
}
```

Listing 2.1: Pseudo code of password derivation

2.3.3 Validation

When a subject wants to access some object with some access right, the capability must be validated. The validation function, see pseudo code example in Listing 2.2, takes three inputs; a capability (E), a capability object (B) and the wanted access rights (AR). First the access rights for E is calculated from E's reduction field, if AR is in the set of E's access rights, the process continues by calling the `derive_password`-function from Listing 2.1, otherwise `ACCESS_DENIED` is returned. With E's reduction field and the object's owner password, another password is derived, this password should match E's password. If these match `ACCESS_GRANTED` is returned, and access is granted for the rights wanted.

```
validate(capability, capability_object, wanted_right)
{
    if (validate_rights(capability->reduction_field, wanted_right))
    {
        calculated_password = derive_password(
            capability->reduction_field,
            capability_object);

        if(capability_password_compare(
            calculated_password,
            capability_object->password))
        {
            return ACCESS_GRANTED;
        }
    }
    return ACCESS_DENIED;
}
```

Listing 2.2: Pseudo code of password validation

2.3.4 Reduction

We will now look into reduction of a capability, this means constructing a new capability with a reduced set of access rights of an existing capability.

Sharing access to objects is very common in access control, however, it should be the case to only share objects with the exact access rights needed. Recall the principle of least privileged.

Therefore it is highly relevant that a subject with a capability can produce a new capability with reduced access rights, and give the new capability to the subject which should have the access rights. This operation is called a *reduction*.

The reduction works as follows, see pseudo code example in Listing 2.3: With the reduction field of the current capability E, we find the first flat sub-field. In that sub-filed the access rights wanted for the new capability is set to ones and the others to zero. When the sub-fields are ANDed together as we saw in Equation 2.1, AR is reduced and will only contain ones where the new access rights were specified. It is important to note that a reduction can only be a subset of the original AR. Since we are changing the first flat sub-filed, this means that if the result of the logical AND operation on the sub-fields before is zero, then it will of course still be zero, no matter if one or zero is placed in a later sub-field.

```
reduction(current_capability, reduced_rights, capability_object)
{
    new_capability = copy(current_capability);
    masked_subfield = first_flat_subfield(new_capability);
    if (not_found(masked_subfield))
    {
        return ERROR;
    }
    masked_subfield = masked_subfield & reduced_rights; // Logical AND

    // Apply next step from the password derivation function
    encryption_buffer = masked_subfield[i];

    // Use password value from the existing capability as key to encrypt
        encryption_buffer
    // and save the new value as password for the new capability
    new_capability->password = encrypt(
        encryption_buffer,
        new_capability->password);

    return new_capability;
}
```

Listing 2.3: Pseudo code of capability reduction

2.3.5 Class and Revocation

As mentioned earlier a known myth about capabilities is difficulties about access revocation [51]. We will now see how e-capabilities solve the problem.

Let's start by introducing the class field. The class field in a capability E is an addition to what we previously have discussed. The class field contains an integer which is the entry in a revocation table RT.

It is the object which maintains RT, an entry in the table can be seen as an extra sub-field, although where the owner of the object can control the values. Changing an entry to all zeroes would mean that all access for that entry, hence class, has been revoked.

With the new class and revocation table effective access rights, EAR, is introduced. EAR is calculated as follows, where RT_c is the class entry in the revocation table.

$$EAR = AR \wedge TR_c \tag{2.2}$$

When a capability is constructed, it has to be specified which class it should belong to, and the revocation table in the object should be updated correspondingly to the access rights wanted for that class. It is a constraint for the revocation table that RT_0 is an entry full of ones, meaning that it grants all permissions. For example E_{own} must be in class 0.

It should also be noted that if a capability is reduced multiple times and remains in the same class, then a change in the revocation table would affect any capability in the class, hence solving one of the difficulties that was pointed out in the myths.

The validation mechanism `validate_rights` used in Listing 2.2 should of course validate with respect to EAR with this addition.

3 Disk Encryption

In this chapter a general introduction to disk encryption is given and in Section 3.2 we will go into debt with Damgaard and Dupont's [21] Disk Encryption Scheme. This scheme is later used as the fundamental design in ACRYLICS.

3.1 Disk Encryption

Disk encryption is one of the strongest tools to protect data and systems [16]. It has been used in many ways and at many levels. The most classical levels are encryption on files, file systems or the whole disk [38].

At file level encryption, only files that have been chosen are encrypted. The files will each have an encryption key and passphrase. This does not scale very well if many files should be encrypted.

At file system level, the file system itself offers encryption, this means that the entire file system is encrypted, hence all files on the file system. It often only requires one passphrase to get the encryption key of the entire file system. We will see examples of such, like CryFS or ZFS, in Section 4.3.

The disk level often offers encryption below the file systems. This means that any data sent to the disk, regardless of the file system, and sometimes regardless of which partition the data are written to, the data will be encrypted. This is also known as Full Disk Encryption (FDE). So the file system itself can be of any type, but the data will be encrypted when it is written to disk, this is of course the main advantage with disk level encryption, since it happens transparent to the system.

It should be noted that it is not always a good thing with transparent disk encryption between the file system and the encrypted disk. This can be a security gap which is addressed later in the analysis in Chapter 4.

There are multiple ways to achieve encryption at disk level. For example tools like BitLocker, FileVault and dm-crypt. Some of them can make use of supported cryptographic hardware. These will also be further analyzed in Chapter 4.

3.2 Theory for Disk Encryption Scheme

In this section we will go in depth with Damgaard and Dupont's [21] Disk Encryption Scheme. The scheme is later used as the core design of ACRYLICS. The scheme defines four players; each has responsibility for a given part of the system. We will follow the article and describe the four players: User, File System (FS), Encryption Module (EM) and Disk.

It is worth to underline that it is a disk encryption scheme, meaning that the File System can be any file system, and it is EM which handles the encryption. For simplicity some technical details are omitted in the introduction of the four players.

First some security notation is introduced, then the four players are introduced and finally we will look at the passive and active security of the scheme.

3.2.1 Security Notation

The security poofs in Damgaard and Dupont [21] are constructed with respect to the UC framework [13], where an IdealDisk (ID) is modelled which does what we try to achieve, and as long as it is not possible for the UC environment to distinguish ID from the real disk in the protocol with the fours players, this proves the security, because an adversary would be unable to learn anything about the content on the disk.

Two important security metrics are passive- and active security. In passive security an adversary can only observe, whereas in active security the adversary is allowed to change things. Specifically in our case this is whether or not the adversary is allowed to directly manipulate data on the physical disk.

Disk encryption is often done with symmetric block encryption. Given a key, symmetric block encryption is a deterministic process and only one block is encrypted at a time. To circumvent this, Mode of Operations are introduced. In [21] symmetric block encryption is used with Cipher Block Chaining (CBC).

With CBC an Initialization Vector (IV) is randomly generated and has the same size as one block. The IV is XORed with the first plaintext block, and the block is encrypted with the given key, and ciphertext is produced. If there are multiple plaintext blocks, the output of the first encrypted block will be used to XOR the next plaintext block before that block is encrypted. This process continues until all blocks are encrypted. As the IV is needed to decrypt the first block in CBC, the IV is a part of the output, hence the output is expanded, and the encryption is no longer deterministic and multiple blocks can be encrypted together.

There exist multiple modes of operation with different approaches. In some of the encryption tools mentioned before it is common to see GCM (Galois/Counter Mode), CCM (counter with cipher block chaining message authentication code) or XTS (XEX-based tweaked-codebook mode with ciphertext stealing).

3.2.2 Model

User

The user can call four commands; *Initialize*, *Boot*, *Read* and *Write*. On *Initialize* which is only called once, the User chooses a password and uses the password to construct some initial data which is sent and written to Disk. On *Boot* the user sends its password to EM, EM responds with accept or reject. On *Read* and *Write* the user asks FS to either read or write data to the disk. On a *Read* either the data or ERROR is returned. On a *Write* either OK or ERROR is returned.

File System (FS)

The File System forwards *Read* and *Write* commands to EM. The File System handles where data is stored in logical sectors and gives EM a list of logical sectors where the data should be written to or read from. EM translates logical sector addresses into physical sector addresses and encrypts or decrypts respectively. This makes the encryption transparent for the File System, since the file system only sees and sends unencrypted data. Since FS only forwards data, it makes no real difference if FS is corrupted. If FS forwards corrupt data, it will come from a corrupt user, which the scheme cannot not defend against.

Encryption Module (EM)

EM handles the encryption and has the encryption key stored in its internal state. EM's only source of random input comes from the user via the password, so if the user is corrupt, no security can be guaranteed. User gives its password to EM on *Boot*, from which the encryption key is derived.

As discussed, EM takes *Read* and *Write* requests from FS and encrypts or decrypts the data depending on *Read* or *Write*. EM then either sends encrypted data to the Disk for storage, or gets encrypted data from the Disk, decrypts it, and returns the decrypted data to FS.

Disk

On *Initialize* from User, Disk writes the initial data computed by User to the disk. When Disk is initialized, it servers *Read* and *Write* requests from EM, and either returns or writes some data to the requested sector.

3.2.3 Passive Security

With passive security we want to ensure that no adversary observing the disk will learn anything about the data.

The first thing observed is that no deterministic encryption algorithm can be passively secure. This means that it is not possible to have a one-to-one correspondence between input and output from the encryption. In the deterministic case the same input will always produce the same output, which gives information away about the input data.

Suppose a deterministic encryption algorithm is used, an adversary could observe a sector and remember the data for that sector. If later the sector is freed and the adversary legally gets assigned the sector, the adversary can get a guess of what the observed data was. More precisely the adversary can choose some plaintext and see what ciphertext ends up at the sector and compare if the new ciphertext is the same as the earlier observed ciphertext, hence a deterministic encryption algorithm cannot be secure.

To get around a one-to-one mapping, the output needs to be expanded and a semantic secure algorithm is needed to obtain passive security. This is done with the CBC mode of operation as introduced in Subsection 3.2.1.

Theorem 1 from [21] uses the UC framework model to prove that any disk encryption scheme as specified is passively secure. This is done by simulating reads and writes with random encrypted data on the ID and real encrypted data in the protocol. Since we use a semantically security encryption algorithm, by definition, it is not possible for the adversary to distinguish the random encrypted data from our real encrypted data.

Furthermore a padded CBC continuation scheme is presented in [21] and proved secure in Theorem 2. However, we will not make use of this particular scheme and will not go into further details. The padded CBC continuation scheme is used as a way to avoid random bit generation on the fly. Since ACRYLICS will demand random generated bit on the fly, this is not relevant for the ACRYLICS design.

3.2.4 Active Security

With active security we want to obtain that no adversary can modify the disk content without it being detectable.

Even though semantic security implies passive security. Semantic security is not enough to obtain active security.

Assume that an adversary observes a user creating a file, which is going to be saved in some sector. The adversary saves the ciphertext C it sees. If the file at a later point is deleted, the adversary hopes that it can create a file and get allocated the same sector. As the adversary can write directly to the disk, the adversary can circumvent the encryption and write the ciphertext C back to that sector. The adversary can now issue a normal read command and get C decrypted by the system. Compared to the passive adversary which could not circumvent the encryption mechanism, the passive adversary could only issue a normal write and get a guess of the data. Whereas the active adversary can trick the system to decrypt the original data.

Hash-Tree

To obtain the goal of active security, we need to prevent the adversary from modifying the disk without being detectable. If we can ensure the integrity of the entire disk, we will be able to check if some sector has been modified without using the four players properly.

The proposed solution is a Hash-Tree containing Message Authentication Codes (MAC), where every node has the MAC for its children and the leaves have the MAC for data-sectors. Some extra sectors are needed to store the tree. The header sector of the extra sectors will contain the MAC of the Hash-Tree's root node, which can be verified at boot. By verifying the root node we know the integrity of the root node is correct. With the integrity of the root node intact, the MACs for its children are used to ensure the integrity of the childrens, and so forth down to the leaves. The leaves have the MAC for the data sectors, and we can therefore validate the integrity of the entire disk if we can ensure the integrity of the root node. If any integrity check fails, this would mean that a sector has been modified without using the four players properly, and it is detected.

Theorem 3 in [21] shows that active security is obtained with the proposed Hash-Tree solution. Theorem 3 also uses the UC framework, ID and the proposed protocol as in Theorem 1. Additionally a simulator is constructed to be an ideal simulator of the proposed Hash-Tree solution. The simulator keeps track of MAC values corresponding *tag*, where *tag* represents a historical version of the disk, and is updated on every write. The simulator behaves much like EM but substitutes random bitstring for CBC mode encryptions. The tricky part here is the boot sequence, where the simulator must be able to provide a *tag* to ID, which EM does not normally do. If the adversary sends a wrong (root, MAC) pair, the simulator must know by checking if the MAC is consistent with the *tag* remembered. If it is consistent then the simulator must be able to provide the consistent *tag* to ID. If it is not consistent, the simulator must replace the request with a malformed package which is rejected by the protocol as normal and the adversary is prevented from changing the disk content.

By a sequence of scenarios in Theorem 3, it is proven that the UC framework environment cannot distinguish the first scenario from the last, unless it can forge a MAC, produce a hash collision or can distinguish chipers from random permutations, hence active security is obtained.

Theorem 4 in [21] shows it is possible to implement additional levels below EM, e.g. caching and journaling, or restructuring layer below EM, where the system will still be secure. It is important that the layering is strict, and EM must for example not know anything about the file system, and otherwise, they must follow the protocol as specified.

The security could be improved by remembering a small string from session to session. As described above the integrity relies on the MAC for the Hash-Tree root node. However the root note could itself be tampered with. Therefore it is suggested that a small string could be generated and remembered from session to session. The string has to be generated on every write, since the MAC changes on every write. On *boot*, the string can be compared to the last generated. The drawback is to get a user to keep track of the last generated string and remember it for next boot.

Security Allowing Some Corruption on EM

So far we have assumed that it is not possible to corrupt EM, but this assumption cannot be made straight forward. It would require a special uncorruptible unit (RO). RO would require some fixed secret inside which would make it possible to supply secret material to EM, hence ensuring that EM cannot be corrupted.

In the real world RO could be compared to the TPM (Trusted Platform Module), which is a cryptographic coprocessor, which also can store keys secret.

However, it cannot be assumed that a system has this unit and therefore it has to be considered whether some corruption of EM can be allowed.

It should be clear that if an adversary can take over the EM completely there will be no security left. In that case the adversary would have access to the encryption keys and can directly use the keys to get the data.

But what if the adversary is allowed to take a snapshot of EM and then has to leave again without being allowed to do any modification. Obviously the adversary gets the keys for the current contents of the disk, leaving the adversary able to decrypt all current content, however, we can prevent the adversary from getting access to any new information on the disk after the snapshot was taken. This is done with the improved key scheme:

Improved Key Scheme

So far only one encryption key has been derived from the user's password, selected initially.

Two things can be improved. Allow the user to change password and use a structure like the Hash-Tree explained in Subsection 3.2.4 containing encryption keys, where the keys change on every encryption.

The users can be allowed to change passwords by generating a public key pair, which are encrypted by the key derived from the user's password. If the user changes password, then the public key pair is encrypted with the newly derived key instead. The public key pair is used to encrypt and decrypt the root node for the hash- and key-tree. The private key is used to decrypt and the public key to encrypt. The private key is only needed in memory at boot time, and then it can be forgotten. The public key is kept in memory such that the root node can be encrypted on every write. Recall that the Hash-Tree is updated on every write.

The Key-Tree functions the same way as the hash-tree, on every write we generate a new random key in each node on the path to the leaf and use it to encrypt its children. This means that on a read, the path is followed to the leaf and the key in a node is used to decrypt the child. When reaching the leaf, the key is used on the corresponding data-sector.

If the adversary has a snapshot of EM, the adversary will only have the keys for the current content; when a new legit write is made, the tree is changed and the adversary cannot know the new keys.

However, the adversary knows the public key and can therefore construct new root nodes and in fact completely change the tree, but the adversary will not be able to boot the system again, since the adversary does not know the password to derive the private key, which ensures that the adversary cannot learn more after the snapshot.

4 Analysis of Currents Systems

In this chapter we will analyze different sets of current state-of-the-art solutions for securing data, systems and users. In Section 4.1 cryptography hardware support is introduced, this is done with respect to the Trusted Platform Module (TPM). In Section 4.2 the most known and relevant operating systems and in particular their security features are analyzed. Notable also Full Disk Encryption (FDE) tools to Windows (BitLocker), Linux (dm-crypt) and macOS (FileVault) are introduced. In Section 4.3 some remarks on file systems are presented, importantly the integrated encryption in ZFS and CryFS. Lastly in Section 4.4 the findings is discuss and **Security Level 1** and **Security Level 2** from the Security Level model are defined, as illustrated in Figure 1.1. The security levels are later used to compare the security of ACRYLICS to the analysed solutions.

4.1 TPM (Trusted Platform Module)

In this section information is found in the TPM specification [75], in following papers [64, 76] and additional information has been found from websites [58, 68].

The TPM-chip is a Secure cryptoprocessor and can be used for different crypto-operations like key- and random-number generation, secure storage of keys and most importantly it is used as the root of trust. TPM 1.2 was the old specification and TPM 2.0 is now the newest standard. The TPM standard is open source, and therefore all information can be found in the TPM documentation.

The TPM is a practical example of the RO unit discussed in Subsection 3.2.4. All newer computers have a TPM chip on the motherboard, except for MacBooks, because Apple uses their own security chips, which are discussed in Subsection 4.2.2.

Let's go into some details about the newest TPM 2.0 specification. The main structure in the TPM 2.0 is build with four hierarchies

- Endorsement Hierarchy
- Storage Hierarchy
- Platform Hierarchy
- Null Hierarchy

The TPM is born with an Endorsement Primary Seed (EPS) which is unique per chip. EPS is burned into the chip and cannot be changed, the EPS resides in the endorsement hierarchy and is used to create the Storage Root Key (SRK) where the private part never will leave the chip and can only be proven to exist by decrypting something that is encrypted with the public part. SRK is used as the root of the entire key hierarchy.

One of the important use cases for TPM is Secure boot. When a system is booting, the TPM is used as the root of trust, and will not allow the system to boot if malicious manipulation is detected. The TPM has 15 Platform Configuration Registers (PCR) which are used to store hashes when the system is booting, hash values from all the hardware devices and hashes of boot firmware code are calculated and only if the hashes match up the system will boot as the integrity of the system is confirmed.

An operating system can take ownership over the TPM, this is done with the creation of a new SRK. A notable limitation is that only one operating system can have ownership over the TPM. So in a dual boot environment only one of the operating systems can use the TPM for key storage.

With TPM 2.0 ”Enhanced Authorization” was introduced, which can be used to construct very sophisticated access control mechanisms [64] which also makes use of some of these PCRs to hold session hash digestives, and only if the final hash match up access are granted. It is a very advanced feature and to limit the scope it has not been considered further, however, it is believed that the ACRYLIC system could make use of this in some scenarios.

Although TPM is used for secure computing, there are some serious concerns about using the chip. There have been different physical attacks where an intruder can read out keys of the TPM. For example BitLocker's disk encryption key can be extracted, hence making it possible for an intruder to decrypt data unauthorized [76, 68, 58].

We should also consider if binding data to hardware can be problematic. If for example the device breaks, then no one can decrypt the data. When backup is made this should also be encrypted, but binding backup data to a single device also seems fragile.

4.2 Current Operating Systems Security Mechanisms

In the section some of the current and most known Operating Systems are analyzed, including a relevant system called seL4. Especially their security features are analyzed and how they protect their systems. Concretely we will in the following subsection look at Windows, Apple's MacOS and iOS, Linux, Android, FreeBSD and seL4.

Most of the information in this section comes from different talks from developers and security researchers, why some of the references are from either talks, blog posts or directly information from vendor home pages. It has been a goal to validate information between vendors web pages with the found literature, which is believed to give the most accurate analysis as possible.

Some systems are proprietary and there exist countless of different features and systems, through general research and available information both on vendors websites and research papers, all largest state-of-the-art solutions are analyzed, and has been investigated and described as far as possible with open litterateur.

4.2.1 Windows

In this subsection Windows is analyzed. Information for this subsection has been found in following papers: [69, 81, 76, 29, 40], vendor web pages: [49, 47, 50] and the 2015 blackhat conference talk ”Battle Of The SKM And IUM: How Windows 10 Rewrites OS Architecture” by Alex Ionescu and the 2016 blackhat conference talk ”Analysis of the Attack Surface of Windows 10 Virtualization-Based Security” by Rafal Wojtczuk.

Secure Boot

Microsoft and Windows have been a huge factor for the direction of development in modern computers. One result is the TPM as just introduced in Section 4.1. It is used together with UEFI firmware and Secure Boot which are also heavily influenced by Microsoft.

Secure boot was introduced when Windows 8 was launched and it is now one of the most important tools for root-of-trust. The secure boot process is as described in Section 4.1.

Some Linux distributions take advantage of Microsoft's work with secure boot and makes it possible to boot some Linux distributions with secure boot enabled. If an operating system does not support secure boot, this has to be disabled in firmware settings and the secure boot process and integrity check will not be made.

As we discussed in Section 3.2.4 it is a very hard task to validate the boot process and currently Secure Boot is the state-of-the-art to address this problem. Apple uses their own hardware module to do something similar which we will analyze in the Apple Subsection 4.2.2.

BitLocker

Furthermore Windows also uses the TPM for their disk encryption tool called BitLocker. BitLocker is not enabled by default and has to be activated by the administrator of the system. Furthermore it should be noted that BitLocker is not included in Windows 10 Home edition. Bitlocker can also be used without a TPM installed, but will require some workarounds, we will focus on the case where a TPM is available.

BitLocker encryption keys are stored inside the TPM. This was validated by a small test where BitLocker was enabled and the TPM was cleared afterwards. On the next boot Windows was not able to decrypt the hard drive due to missing encryption keys. Although it requires physical access to the computer, this seems like a warning flag. It only takes a few minutes to boot the computer and clear the TPM in the firmware settings. In the default case, this requires no passwords or security checks to do at all.

The data was however not totally lost since a unique 48-digit numerical password (recovery key) was made due the BitLocker enabling process, but only if the user has this key written down or stored somehow remotely, then this key can be entered and can recover the original encryption keys and the hard drive can be decrypted.

Virtualization Based Security

We will now move our focus to the architecture of Windows and which security features they have introduced in the architecture. Microsoft's current keystone for security is Virtualization Based Security (VBS). Although this is only for Windows Enterprise and Server (2016 and newer).

Let's start by clarifying the terms Secure Kernel Mode (SKM) and Isolated User Mode (IUM). The idea with VBS is to set up Virtual Secure Mode (VMS) to create two virtual levels; Virtual Trust Level 0 (VTL 0) and Virtual Trust Level 1 (VTL 1). These levels can be seen as virtual machines, hence the Windows Hypervisor is required. The higher a VTL is, the more privileged it is. This means VTL 0 is *normal mode* and VTL 1 the *Secure mode*. This should not directly be transferred to the Ring Levels of the CPU, where ring 0 is kernel mode and ring 3 is user mode. In each VTL the CPU can be in any ring, this makes kernel and user mode possible in both VTLs.

The VMS is the hypervisor-facility which makes the features Device Guard, Credential Guard, virtual TPMs and shielded VMs available. We will not go into specific details with all the features, but we should get the understanding of how the structure works. The mentioned features are living in VTL 1 - the secure mode. From VTL 0 we want to make secure requests to the mentioned security features. For example let's consider Figure 4.1. If we imagine that we want to authenticate for some operation (Credential Guard), we will execute a command from VTL 0 and CPU Ring level 3, this is the normal mode's user mode. The authentication call is done by a syscall to the normal mode's kernel mode (CPU Ring level 0). Via the hypervisor the request enters the Secure Kernel (SKM) in secure mode (CPU Ring level 0), SKM hand off the request to the authentication process running in IUM, which is running in CPU Ring Level 3, limiting the privileges for the secure processes can do, hence greater security.

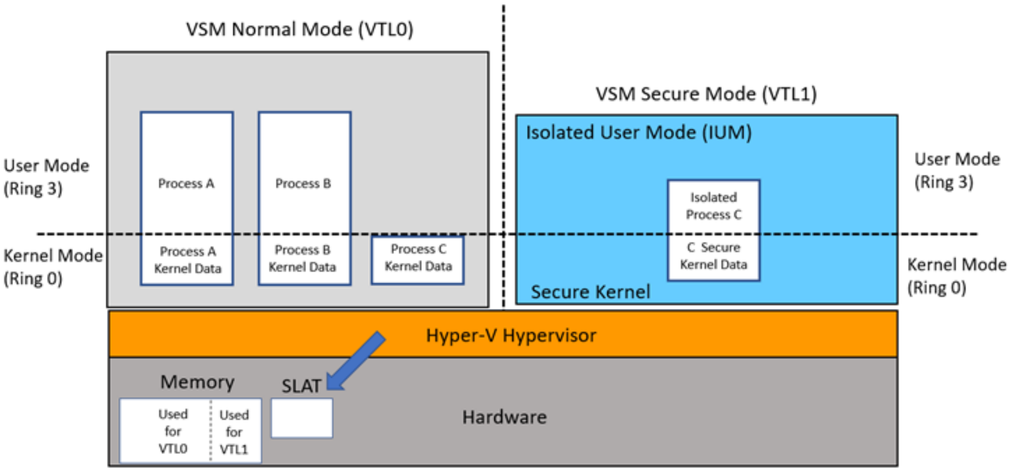


Figure 4.1: Windows' Credential Guard architecture illustration from https://docs.microsoft.com/en-us/windows/win32/procthread/isolated-user-mode--ium--processes, Accessed: 2021-05-31

Note the highlighted SLAT on the figure. This is the Second Level Address translation, this is the unit translating addresses from VTL 0 to VTL 1 and vice versa, this prevents VTL 0's ability to directly access things in VTL 1 and gives the power to control the information flow.

Furthermore, processes in IMU are called trustlets and are isolated processes with a strict syscall interface to SKM.

For the VBS and VSM to be secure it is highly important that VTL 1 is not compromised. The point of the architecture is to ensure that if something happens in VTL 0, even if the kernel in VTL 0 is compromised, then VTL 1 will prevent further compromising. VTL 1 can for example know cryptographic secrets which for example can be used to prevent an intruder that want to make privileged escalation (authentication request as discussed) or execute malicious software which are not signed (using another feature called Hypervisor-Protected Code Integrity (HVCI)).

To ensure that VTL 1 is not compromised, the Secure Boot process is used to integrity check the whole system as discussed. If the Secure Boot process succeeds VTL 1 is trusted, and can perform the trusted operations that are wanted. However, it is possible to run the setup without Secure Boot enabled, but this is leaving the security gap that it cannot be guaranteed that VTL 1 has not been tampered with.

Default Security Windows Pro Installation

As mentioned VBS and VSM requires Windows 10 Enterprise or Windows Server 2016 or higher and also requires advanced setup. With the analysis we also want to know which security there is for standard users, therefore we want to analyze the security of a standard Windows Pro consumer installation.

Firstly it is noted that any advanced security feature is disabled by default. Examples seen in figures 4.2, 4.4 and 4.8.

BitLocker in the Wild

Recall that BitLocker is not installed by default in Windows Home edition and in pro edition it is not enabled by default, this is seen in Figure 4.2.

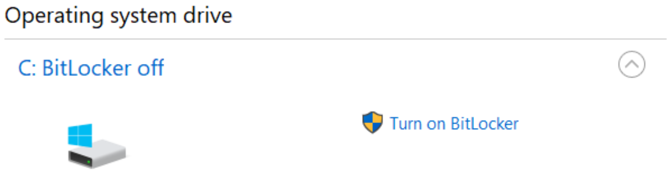
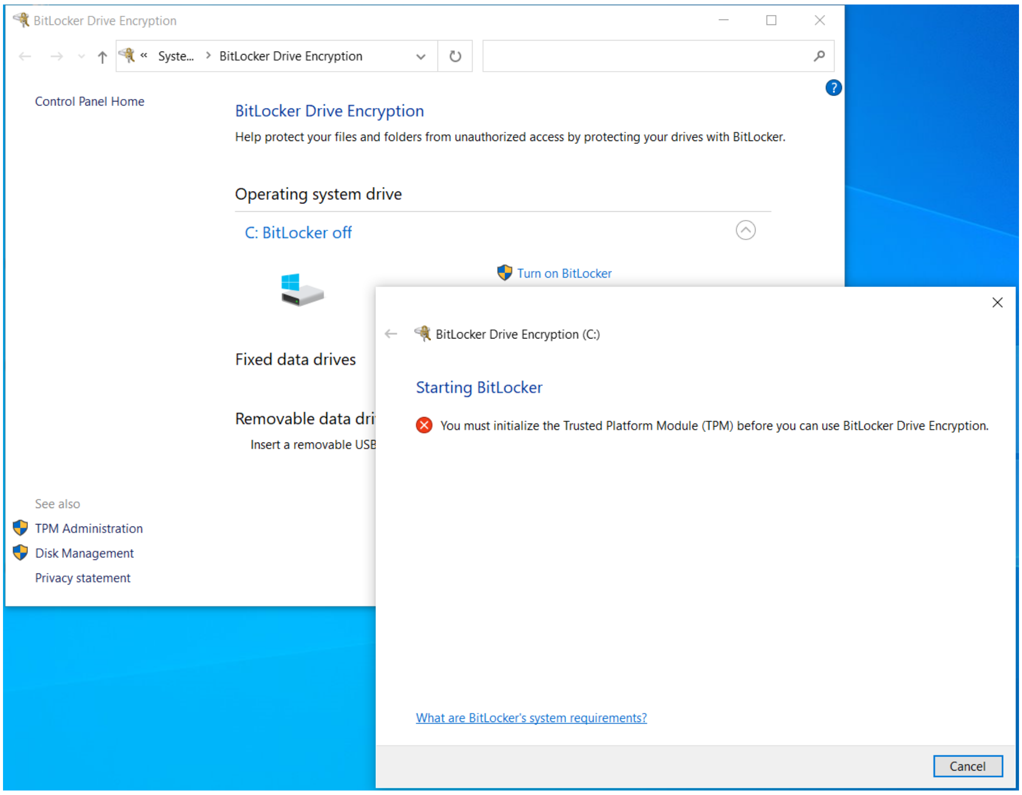


Figure 4.2: Screenshot of Windows showing BitLocker disabled by default in Windows Pro

To confirm about BitLocker usage of TPM and TPM ownership in general, Windows does not have the ownership of the TPM in the analysis test, and Windows cannot use it to store encryption keys. In Figure 4.3 enabling BitLocker is tried and we see that windows says that the TPM need to be initialized, hence letting Windows take the ownership and generate a new RSK (Root Storage Key) as described in Section 4.1. It should be noted that BitLocker does not rely on VBS.



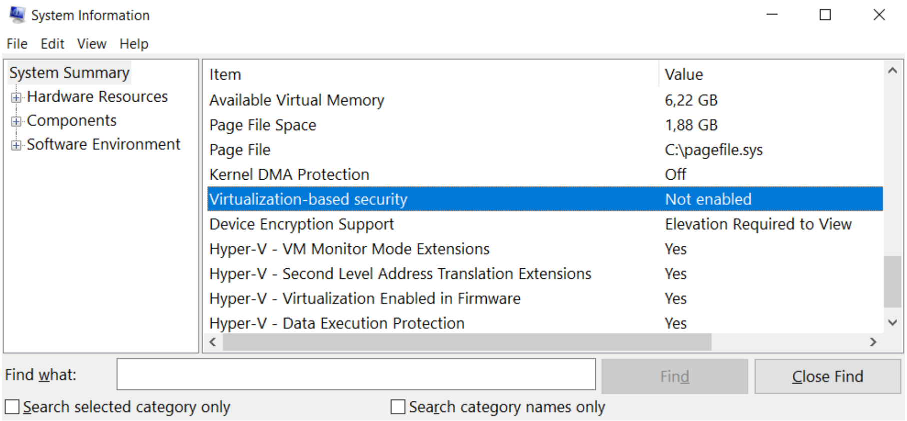
Help protect your files and folders from unauthorized access by protecting your drives with BitLocker.

Insert a removable USB

You must initialize the Trusted Platform Module (TPM) before you can use BitLocker Drive Encryption.

Figure 4.3: Screenshot of Windows showing the case where Windows does not has the ownership of the TPM and says that it needs to be initialized before it is possible to enable Bitlocker

Even though the current understating is that VBS should only be a part of the enterprise and server editions, there is still an entry in the *system information*, see Figure 4.4. It is not entirely clear from open literature if it is only the VSM-part that is for enterprise and server editions or exactly which features are accessible or not in which editions.



Item	Value
Available Virtual Memory	6,22 GB
Page File Space	1,88 GB
Page File	C:\pagefile.sys
Kernel DMA Protection	Off
Virtualization-based security	Not enabled
Device Encryption Support	Elevation Required to View
Hyper-V - VM Monitor Mode Extensions	Yes
Hyper-V - Second Level Address Translation Extensions	Yes
Hyper-V - Virtualization Enabled in Firmware	Yes
Hyper-V - Data Execution Protection	Yes

Find what:

Find

Close Find

□ Search selected category only

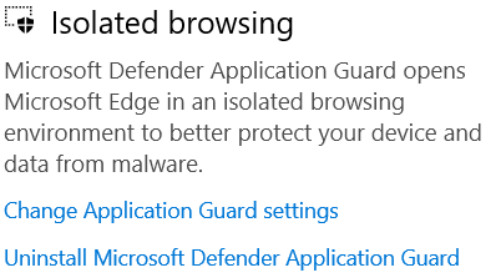
□ Search category names only

Figure 4.4: Screenshot of Windows showing VBS is found, but not enabled

Application Guard and Isolated Browsing

However, it is possible to enable and install Application Guard and enable Isolated Browsing. Application Guard is a feature which allows Edge to run in a virtual environment, this means that if some malware tries to exploit Edge when a user is surfing websites online, it would not have access to the rest of the system.

The installation is done through *Windows Features*, see Figure 4.6, and will require a reboot of the system. However, there are also some unclear things here. In Figure 4.6 we can see that Hyper-V is not installed. Recall this was required for VBS and VMS to run, but on the analysis test system this is not installed, however VBS is enabled after the reboot, see Figure 4.7



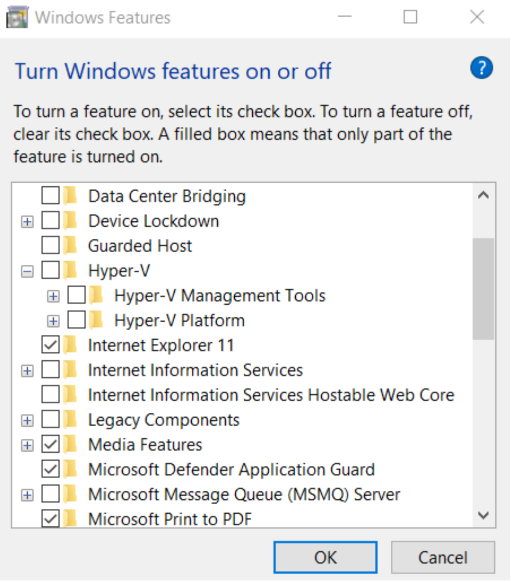
Isolated browsing

Microsoft Defender Application Guard opens Microsoft Edge in an isolated browsing environment to better protect your device and data from malware.

Change Application Guard settings

Uninstall Microsoft Defender Application Guard

Figure 4.5: Screenshot of windows showing the Application Guard Settings



Windows Features

Turn Windows features on or off

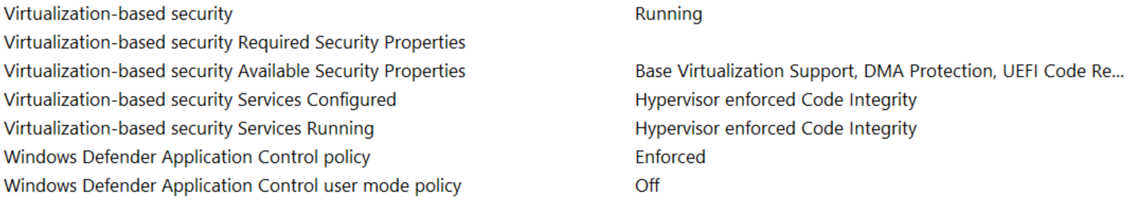
To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- □ Data Center Bridging
- □ Device Lockdown
- □ Guarded Host
- □ Hyper-V
  - □ Hyper-V Management Tools
  - □ Hyper-V Platform
- ☑ Internet Explorer 11
- □ Internet Information Services
- □ Internet Information Services Hostable Web Core
- □ Legacy Components
- ☑ Media Features
- ☑ Microsoft Defender Application Guard
- □ Microsoft Message Queue (MSMQ) Server
- ☑ Microsoft Print to PDF

OK Cancel

Figure 4.6: Screenshot of Windows showing the Windows Features it is possible to enable a lot of features, in particular Application Guard

So at least some of these features can be enabled in Windows Pro edition. This should at least settle the point about advanced configuration, and that it is not easy to see and understand what it actually done, and if some configuration is still missing for the features to work correctly. At least non-technical persons would not be able to configure this by themselves.



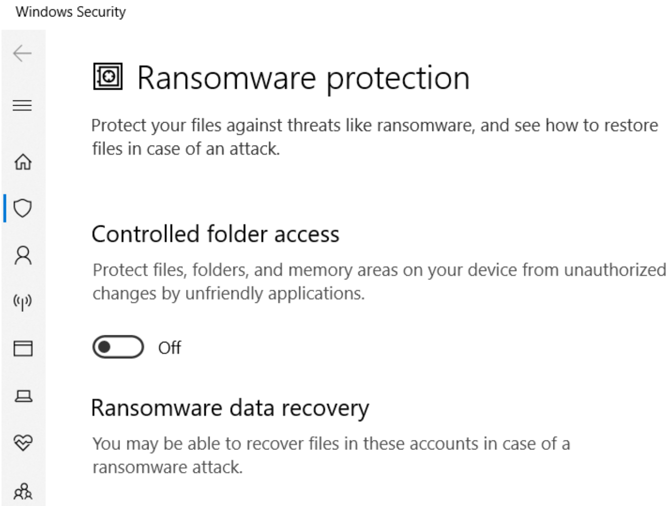
Virtualization-based security	Running
Virtualization-based security Required Security Properties	
Virtualization-based security Available Security Properties	Base Virtualization Support, DMA Protection, UEFI Code Re...
Virtualization-based security Services Configured	Hypervisor enforced Code Integrity
Virtualization-based security Services Running	Hypervisor enforced Code Integrity
Windows Defender Application Control policy	Enforced
Windows Defender Application Control user mode policy	Off

Figure 4.7: Screenshot of Windows showing VBS is enabled after the installation of Application Guard

Controlled Folder Access

The last findings we will discuss is the ”Controlled Folder access”, see Figure 4.8, this is maybe the most directly relevant findings with respect to the later presented ACRYLICS solution. Controlled Folder access does not rely on VBS.

With Controlled Folder access enabled, Windows will prompt the user if some program tries to access something in a folder that it should not. The user can then grant or deny access.



Windows Security

Ransomware protection

Protect your files against threats like ransomware, and see how to restore files in case of an attack.

Controlled folder access

Protect files, folders, and memory areas on your device from unauthorized changes by unfriendly applications.

Off

Ransomware data recovery

You may be able to recover files in these accounts in case of a ransomware attack.

Figure 4.8: Screenshot of Windows showing Controlled Folder Access settings, also disabled by default

The solution aims to prevent ransomware from accessing all files on the system, and if an attack is ongoing then the user explicitly has to press access granted to allow the ransomware to access the files, but as with the other security features, this is optional, and is not enforced. It is also disabled by default.

4.2.2 Apple - MacOS and iOS

In this subsection information has been found in following papers: [67, 2, 57, 72, 34, 19, 22], following pages at Apple’s website [7, 5, 6, 63, 1], following blogspots: [20, 4, 62] and the 2019 Blackhat conference talk ”Inside the Apple T2” by Mikhail Davidov and Jeremy Erickson and from the 2016 BlackHat conference talk ”Demystifying the Secure Enclave Processor”.

Brief History and Background

When investigating Apple, there is a lot in common for the security in their Operating Systems. The main focus will be the security in the macOS for their MacBooks and iOS for their iPhones, but this will also include mentioning watchOS, BridgeOS and SepOS, and this is why it all comes together in a general analysis.

Before diving into the newest security specifics, it is worth mentioning that Apple also has a long history like Windows' and Linux', which was briefly described in the introduction. MacOS is based on the Mach kernel (First version 1985) and BSD (First version 1977). BSD is derived from the original Unix System (1970) and macOS and iOS are therefore also Unix-like as we saw with Linux.

Even though the state-of-the-art solutions we are going to analyze are advanced technologies, it is still important to remember the roots of the systems. Behind all these new technologies is still the same core system, and it is here we will observe the limitations and flaws.

Apple's is going in the direction of hardware security and especially with their Secure Enclave processor and T2-chip. They want to ensure the root of trust and integrity in even more advanced ways than we saw with Windows and TPM. All of Apple's operating systems (iOS, iPadOS, macOS, watchOS or tvOS) are using hardware components to improve the system security. iPadOS, tvOS and watchOS will not be investigated further, but with current information, we should believe that the idea of security in these products follows what we will see in iOS and macOS.

Secure Enclave processor

All newer Apple devices are equipped with the Secure Enclave Processor (SEP) which is a secure coprocessor launched with the iPhone 5s generation. MacBooks are special and have a chip called the T1-chip (Launched with the 2016 MacBook) or the newer T2-chip (Launched with the 2018 MacBook) which are also security chips which includes the SEP.

What Windows is doing with Secure Boot and TPM, Apple is doing this with the Secure Enclave processor. The whole point is to have a very little part that can be verified and be secure from the rest of the system, and let that little part check the integrity of the rest of the system, hence ensure the security of the entire system. Apple is also very sensitive about the update procedure of both apps and the operating system. The whole Apple environment requires signed software packages when updated or installed. Something where SEP also plays a role ensuring constant integrity.

We should distinguish between devices with T2-chip and non-T2-chip devices and it seems that we have to focus on two different operating systems; SepOS and BridgeOS. SepOS is officially described by Apple and is the operating system running on SEP. BridgeOS is the operating system running on the T2-chip, but no Apple-official information has been found for BridgeOS and it is therefore very hard to validate the exact relationship between all these components. However, a lot of reverse engineering results give some strong pointers for the missing information, and the following analysis is based on the combined findings.

SepOS

SepOS is an Apple-customized version of the L4 microkernel and is the operating system running on SEP. L4 microkernels allow formal verification and Apple can thereby sign, verify and certify SepOS and SEP. This finding is quite interesting, since it was already planned to include the seL4 microkernel later in this chapter. seL4 is also based on the L4 microkernel, but is built as a capability-based microkernel.

From the 2016 BlackHat conference talk ”Demystifying the Secure Enclave Processor” more detailed information about SepOS was found, but we will not go into further details. Instead we will turn our view to the T2-chip.

T2-chip and BridgeOS

T2-chip is Apple’s newest state-of-the-art security chip and it is actually more than just a co-processor, it is a 64-bit ARMv8 processor, it is stated that it is not exactly the same as the A10 processor found in iPhone 7 and 7 Plus, but is build from the same core element as the A10-chips.

This means that the T2-Chip is a full SoC (System on Chips) with its own Operating System called BridgeOS, which is a derivative of watchOS. The T1-chip, the first version of the security chip, was based on a processor which was very similar to the processor in the Apple Watch, why BridgeOS is based on the watchOS.

The T2-chip makes hardware support for encrypting the disk available. It has burned in a secured AES 256-bit key into the chip which is used for the encryption and decryption.

For our point of view we should note that when the Mac is booting up, the decryption happens automatically and transparently for macOS. This is of course convenient, but one should be aware that the data is decrypted without user activity.

Apple states that one should activate FileVault, which is Apple’s disk encryption utility. FileVault existed before the T2-chip era, and provides the possibility to enter a user password to derive encryption keys and has to be entered before encryption and decryption can happen. FileVault can now make use of the T2-chip, and if FileVault is enabled, password is required before the T2-encryption can happen, this secures the system in that user action is required before decryption can happen.

As discussed with the TPMs, hardware based encryption also have some drawbacks, mainly about backing up data. If the T2-chips get destroyed somehow, it will no longer be possible to get the burned in encryption key on the 2T-chip, which leaves the data unrecoverable. Apple therefore suggests backing up the data in another way. For example with FileVault activated and TimeMachine to do the backup. In this way it is possible to backup the data encrypted, but without the involvement of the T2-chip. The backup data should also should be encrypted, otherwise the data would just be unprotected in the backup storage.

FileVault

A few more details about FileVault. FileVault can work with multiple users, where each user can have their own password. The credentials are stored and when the system is booting it will prompt for credentials, if access granted the volume master key will be derived.

Since all users derive the same volume master key, it means that no matter which user boots the system, all data can be decrypted. If someone has root access to the file system, then any data can be read.

Pointer Authentication Mechanism

Another advanced security feature Apple is using is a Pointer authentication mechanism, where the highest bits in each pointer is signed with a key, and only if the correct key is applied, then the pointer can be authenticated, and a valid pointer is calculated. It ensures that potential malicious programs cannot just follow pointers and do harm, they need a key to do that.

App Capabilities

Apple is also concerned about capabilities for apps, where developers can ask for permission for different functionality. The end user then has to accept if the app is allowed to get the access. It's a well known scenario on the iPhone where one has to allow access for contacts, Bluetooth etc.

More recent updates of iOS, but also macOS, requires the user to allow different program access to different files.

However as mentioned earlier Apple is very sensitive with software packages and app development. For example it requires a paid Apple developer account to access all these capabilities. It can of course be seen as a good thing, since developers at least have to spend some money to develop an app which may require sensitive capabilities. However, it also seems as a drawback limiting usage of the system, and preventing some developers from making secure apps, and should security really consist of whether intruders want to spend some money first and register some persons information?

iOracle - Jailbreaks, Exploiting the legacy core

The iOracle tool [22] is used to build a graph as complete as possible for access permissions in the entire iOS operating system. Later queries can be made in the graph to see which parts have which permission to what.

In the iOracle paper [22] examples of exploiting the system core are presented, commonly known as jailbreaking or ”rooting the device”, which means an exploit which makes root access possible. Early jailbreak attempts could exploit the device before it was booted, but with the new hardware security in the boot stage, this is now less feasible. Newer jailbreaks are therefore more feasible when the device is running.

One jailbreak used a vulnerability in the classical POSIX symlink command and could link a file to another directory with greater permission, and in the end root access was obtained.

This example shows that adding multiple security layers around the core, does not make the core secure. Root was obtained with the standard UNIX permission bits and shows that the real core design is still sensitive, and if someone breaks the new features, then the complete system may be broken.

4.2.3 Linux

In this subsection Linux' security features are analyzed. Information used in this subsection has been found in the following papers [23, 11] and following blog post [56].

Linux has multiple security features, but no real strict strategy to obtain greater security as we have seen with Windows and Apple. Linux' open source nature makes it possible to come up with many different security features, but as Linux has roots in UNIX where there only were two privileged levels; super user and regular user, this still has dominant effect on the system, and the core system design is making it difficult to enforce the new security features.

Linux Capabilities

Linux Capabilities was introduced in the Linux kernel version 2.2. The idea is to take all the super users capabilities and divide them into smaller capabilities. These capabilities can be assigned individually to a program.

An example is the `ping` command, which must have the `CAP_NET_RAW` capability to work. With `setcap` the capability can be given to `ping` and an unprivileged user can now execute `ping` with `CAP_NET_RAW` without super user permissions. In fact any user can now execute the program with the newly assigned capability. The capability is stored with the file in the Extended File Attributes called `xattr` and cannot really be stored with the users, recall the ACL vs Capability example in Section 2.2.

This also means that the system needs a file system that supports `xattr`'s, otherwise is it not possible to save the information. `xattr` is space for some extra meta data for a file, but if a file is copied to a file system which does not support `xattr`, then the file would lose the information, hence its capabilities. It is also a design decision in Linux that capabilities in `xattr` is *not* copied on a copy.

A larger problem with Linux Capabilities, is that a single capability itself can provide enough power to become a full privileged user, this really ruins the capability idea.

For example the `CAP_SETUID`-capability where a user simply can change its `UID` to 0, which is the `UID` for root, and the user has now root-access.

The suid- and sgid-bit Problem

A long going problem with security in Linux is the suid- and sgid-bits. If these bits are set for a program, it tells the kernel to execute the program with the permission of either the owner's uid or the owner group's gid.

Lets look at the terminal utility `screen` as an example. `screen` allows multi-user-terminals in one terminal, also called a terminal multiplexer. `screen` requires root to access data belonging to the `utmp`-group, which allows `screen` to access multiple users' files.

On a basic Arch Linux system the standard UNIX permission bits for `screen` is

```
(From Linux Arch 5.11.11-arch1-1)
-rwsr-xr-x  1 root root        486248 Mar  4 17:59 screen-4.8.0
```

The `s` in the owners permission bit is indicating that the SUID is enabled, and if executed, it is executed as the owner and not as the user executes it. This means that this particular program is running as root, no matter how it starts.

Developers are ongoing trying to rewrite programs and procedures to get rid of these legacy bits. The next example is from a Debian 9 (Stretch) system.

```
(From Debian 9 Stretch)
-rwxr-sr-x  1 root utmp       457608 Feb 10 17:03 screen
```

In this example we see `s` in the group's permission bit, this means when a user are executing `screen`, then it will be run with that group's permissions. So now the program will not run as root, but will have some permission from the `utmp`-group.

In a Ubuntu 20.04 system, which is newer than Debian 9, we will see that the development has succeeded by removing both suid and sgid-bit completely.

```
(From Ubuntu 20.04)
-rwxr-xr-x  1 root root       474280 Feb 23 18:46 screen
```

The program can now be executed as the user itself and the user's group. However, this raises the question, what about the things that was needed to be accessed in the `utmp`-group?

The multi-user support has simply been removed as default. So when a user executes ”screen”, it cannot access data for other users. This is of course a security benefit, but it is also a trade-off of usability. If multi-user functionality is wanted, then serial steps are required and most importantly the suid-bit is once again required to be set.

It is possible to protect against these bits as UNIX file systems, like NFS and ext4, can be mounted with a ”nosuid”-flag, this tells the running Linux system to block any operation of suid- and sgid-bits. It will only prevent the use of the bits but will not fix the problem.

PAM (Pluggable Authentication Modules)

Another feature in Linux is `PAM` (Pluggable Authentication Modules). It is a way to set up sophisticated access control rules to programs. A program can have a specific file specifying the PAM access rules for that program, normally located in `/etc/pam.d`.

The rules uses some `PAM`-shared object libraries, where each object is one simple rule, binding these together one can make really sophisticated rules.

However, non-technical users might not be familiar with `PAM` and the rules are hard to configure and require deep knowledge about the authentication control and how to integrate them in the systems and programs.

MAC (Mandatory Access Control)

As discussed in Section 2.2 most of the Linux distributions are using DAC (Discrete Access Control) but with MAC (Mandatory Access Control) security can be improved. There are some patches to Linux, which takes MAC into Linux. Two of the most knows are `SELinux` (Security-Enhanced Linux) and AppAmor.

AppAmor is a service which makes sure that applications that should be ”enforced” do not access something that is not allowed. However, it is an add-on to the existing Linux, so even if Ubuntu and other systems come with this, it can be disabled and circumvented.

SELinux is the other solution and is using labels as described in Section 2.2. It was originally developed by NSA and is now developed by RedHat. SELinux is integrated by default in for example RedHat Linux Enterprise, CentOS and Fedora.

It is a very advanced system and provides good security when the system is running, but it can be turned off like AppAmor leaving the standard unix permission back. If SELinux is enabled the legacy Unix permission bit is ignored.

SELinux neither enforces protection of the file system. So if the file system is mounted on another operating system, it is possible to just mount and traverse all files like no security was there. SELinux, and also AppArmor, are also requiring `xattrs` in the file system to store the security labels, and therefore has the same limitation about file systems as we saw with Linux Capabilities.

dm-crypt - LUKS

There exist multiple solution to disk encryption on Linux. We will look at the ”dm-crypt - LUKS” combination.

LUKS (Linux Unified Key Setup) is a disk encryption specification, which specifies an on-disk format and can be used with different frontend encryption systems. In this setting it is the dm-crypt subsystem that is used, it has a ”Device Mapper” interface and has been a part of the Linux kernel since versions 2.6. dm-crypt is used to map the LUKS partition and encrypt and decrypt data in block device fashion and write/reads to the LUKS partition.

This solution does not either support multiple user keys and rely on the same master key to encrypt and decrypt the whole partition. So far it has not been possible to find a solution on Linux where this is possible. Only workarounds where each user has an encrypted overlay file system on the main file system has been seen as a solution for this.

Setup of ”dm-crypt - LUKS” also requires some technical skills and does not work straight out of the box.

4.2.4 Android

In this subsection Android is analyzed. Information in this subsection has been found in following paper [35], following webpages, blog posts and talks [65, 70, 77, 8], and the 2017 blackhat conference talk ”Honey I Shrunk the Attack Surface – Adventures in Android Security Hardening”

Background

Android is a mobile operating system and because of the huge number of mobile devices running Android, it is the most widespread operating system in the world. Android uses Linux as a kernel and has done a lot of work to harden userspace and reduce userspace’s ability to reach the kernel unattended or unauthorized. We will focus on how they secure the OS level, including securing the interprocess communication, how they do app sandboxing, and lastly the BigMAC-tool [35] is introduced and used for further analysis.

Figure 4.9 shows an overview of Android’s architecture. There are multiple layers from System Apps to the Linux kernel in the bottom. Android wants to be the most secure and usable operating system for mobile platforms by repurposing traditional operating system security controls.

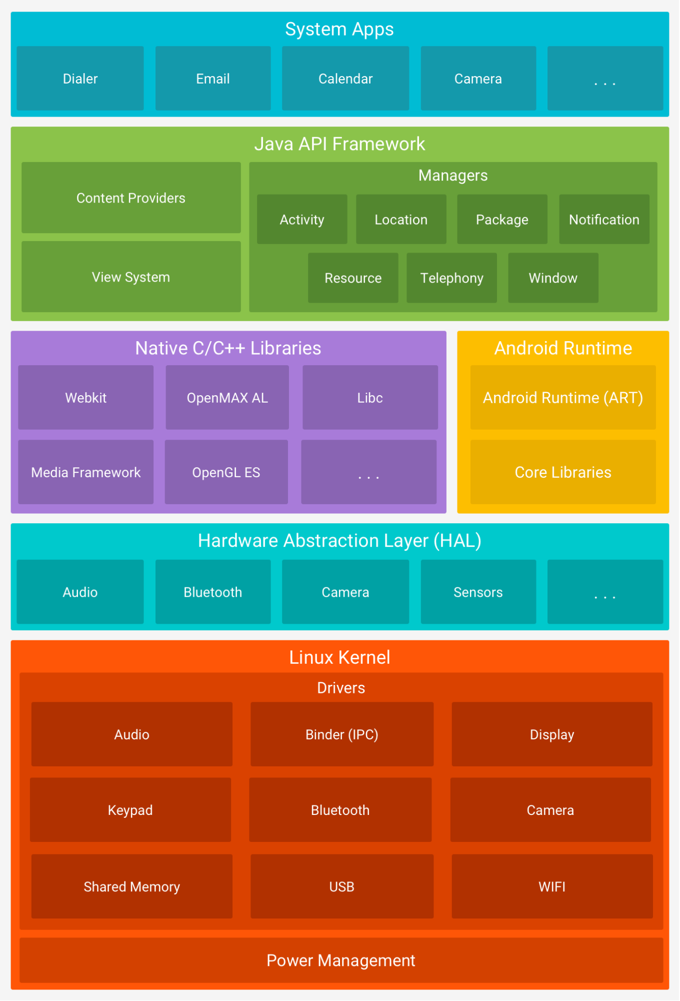


Figure 4.9: Illustration of the software stack of Android from https://developer.android.com/guide/platform, Accessed: 2021-05-31

Android's concrete security goals is:

- Protect app and user data
- Protect system resources (including the network)
- Provide app isolation from the system, other apps, and from the user

Android provides following security features to reach these goals:

- Robust security at the OS level through the Linux kernel
- Secure interprocess communication
- Mandatory app sandbox for all apps
- App signing
- App-defined and user-granted permissions

From a blog post [65], one of the software engineers from Android security, is stating that "Defence is all about increasing the attack chain length and making each link individually harder to exploit". The philosophy of the proposed ACRYLICS system disagrees. It is much more powerful, making the soft vulnerable parts more secure and being able to prove security mechanisms here. Even though a long path has to be traversed to reach the inner Linux kernel in Android, recall Figure 4.9, only one intruder has to breach it once, and then everyone else can use the exploit. Security should come from the core design, not by multiple obstacles. It reminds of the three pigs and the wolf with houses built of sticks or bricks, making a fence around the houses does not make the house more secure, no matter how strong the fence is.

Securing the kernel and interprocess communication

Android's security model heavily relies on a self developed, modified version of SELinux (Security-Enhanced Linux). SELinux enforces MAC (mandatory access control) for files and processes as discussed in Subsection 4.2.3. MAC enforces greater security and introducing SELinux in Android has a huge effect on the kernel security. It should be recalled that SELinux is an optional add-on rather than a requirement, but since Android very much can control their platform, they can enforce the use of it. In the **BigMAC and further analysis** section we will see examples of kernel security issues besides these mechanisms.

Further investigation by Android researches showed that the greatest causes for reaching the kernel from user space was through `ioctl`-calls (interprocess communication). By using SELinux it is possible to restrict access with extended permission rules (XpermRules). XpermRules are a facility in SELinux where very sophisticated access rules can be constructed and restrict access on a fine grained level.

App-isolation

The basic idea for app-isolation in android, is to use the Linux user-based protection. This means that every app assigns a unique user ID (UID) to each app and runs it in its own process. This isolates apps from each other and protects apps and the system from malicious apps and also separate file access between the app's. Since this separation in down to the UNIX permission this ensures isolation down to what they call "kernel-level Application Sandbox".

BigMAC and further analysis

BigMac [35] is a tool for Android like iOracle is for iOS as discussed in Subsection 4.2.2.

BigMac is also building a graph over permission, and it is possible to query which files and processes that have which rights. Discussions in [35] confirms a lot of things already discussed. Firstly we should note that Android is using the classical UNIX permission and the standard DAC system we already have discussed in Subsection 4.2.3. Android also uses MAC in terms of SELinux and lastly Android also uses Linux Capabilities, which we discussed in Subsection 4.2.3. This means that Android uses almost all permission systems in Linux.

There are most likely arguments for and against this, but at least the complexity of the complete access control system is enormous and certainly complex to verify and BigMAC tool tries to enlighten this task.

Recall that Linux Capabilities had the disadvantages, that one single capability can be so powerful, that it can be used to obtain all other capabilities. This is also the case in Android. In [35] a query search for processes with the `CAP_SYS_ADMIN` resulted in a finding of 25 daemons that had this capability and could be used to achieve privilege escalation like the privilege escalation attack reported in CVE-2018-9488.

Another example in [35] is about the `system_server`, where the `system_server` module has the capability `CAP_SYS_MODULE`, which allows `system_server` to load arbitrary kernel modules, and it is concluded that `system_server` must be refactored into smaller services, to break up privileges more fine grained.

In [35] the lack of granularity of `CAP_SYS_ADMIN` is also highlighting and the Linux capabilities in general. Concretely in Android, any process with the `CAP_SYS_ADMIN` capability can get arbitrary code execution in any other domain it can transition, which is a sign of weakness of the capability security model itself.

Combining all these security control systems is very complex and it is not obvious when only analyzing one of the systems in isolation that problems can arise. In the end it all comes down to these classic security problems from legacy UNIX design choices, which cannot compete with the security requirements today. There is missing one canonical and central way to obtain security and the argument that defence is all about creating a longer and harder chain to the critical parts seems to be a bad assumption.

4.2.5 FreeBSD

FreeBSD is also a UNIX-like operating system and has the same difficulties with legacy UNIX features as already discussed throughout this chapter. One could note that FreeBSD offers OpenPAM in contrast to Linux PAM, but the idea is the same and not so interesting for further analysis. More interesting is FreeBSD's capability and sandbox framework Capsicum [79]. Capsicum was started to be integrated into the Linux kernel, but the last code commit was on Sep 21, 2017. The repository is Google's repository and is now marked as unmaintained and Capsicum has never integrated into Linux [15].

Let's examine how Capsicum can be used in FreeBSD. A test program has been constructed which will be used for the examination, see Listing 4.1.

```
#include <sys/capsicum.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
int
main(int argc, char *argv[])
{
  int c;
  int errs;
  int fp;

  char *buf = "Write allowed by Capsicum\n";
  cap_rights_t r; // Current granted rights

  errs = errno;

  // If cap_enter() is called before the file is openen,
  // it will not be allowed to open the file
  // cap_enter();
  fp=open("test.txt", O_WRONLY);

  if (fp < 0 )
  {
    printf("File is not opened\n");
    return 0;
  }

  if ((c = cap_enter()) != 0)  {
    fprintf(stderr, "cap_enter failed: %s\n", strerror(errno));
    errno = errs;
    return c;
  }

  // If the Write Capabiltity it not granted
  // the program cannot write to the file
  cap_rights_init(&r, CAP_WRITE); // Grant write capability
  if ((c = cap_rights_limit(fp, &r)) != 0) {
    fprintf(stderr, "cap_rights_limit failed: %s\n", strerror(errno));
    errno = errs;
    return c;
  }

  write(fp, buf, strlen(buf));

  return 0;
}
```

Listing 4.1: Capsicum test program

Capsicum is used to limit a program's capabilities and use the principle of least privileged. The whole Capsicum environment start with the `cap_enter()`-call. The Capsicum interface is include via `#include <sys/capsicum.h>`.

After the entry call the program can be considered to be in the sandboxed capability mode and the program is only allowed to issue system calls operating on file descriptors or reading limited global system state, this means that any file that the program should access has to be opened **before** the entry call. If a file is attempted to be opened after the entry call, it will fail.

If the program has opened the file before the entry call, Capsicum can be told what the program should be allowed to do on file descriptor. As seen in the test program (Line 38) we assign the write capability to the program, such that it is allowed to write to files, this is done with `cap_rights_init(&r, CAP_WRITE);`.

When the capability has been granted, Capsicum should of course then check if the program has the required capability when a specific action is about to be performed. In our test program we want to write to the opened file. The permission check is done with `if ((c = cap_rights_limit(fp, &r)) == 0)` (Line 39). If the check fails, permission is denied and some error handling should be done. Otherwise it is confirmed that the program has write permission and a normal write call can be used.

For the first time we see an example of programming with capabilities in source code, and the reasoning about Capsicum seems really good, however there are some pitfalls which we will dive into. The use of Capsicum is optional and can be circumvented. Only executables which are programmed with Capsicum and follow the procedure correctly are benefiting from it. There is no system requirement that enforces the use of Capsicum. Even in the test program the checks can circumvent by avoiding the check call, e.g. `if ((c = cap_rights_limit(fp, &r)) == 0)` (Line 39).

If a file is opened before the Capsicum entry call, the program has all the permission to the file that the file is opened with and what UNIX permission allows. If the Capsicum entry call is called, the program will still be sandboxed and it is not possible to open new files and the program are restricted with system calls, but it is the programmers responsibility to do the `cap_rights_limit`-checks on open files. Even a programmer that has all good intentions can forget these checks, which can lead to vulnerabilities opening and this is not optimal.

4.2.6 seL4 - Microkernel

seL4 is an example of an active state-of-the-art capability-based system [33]. seL4 is a microkernel in the L4 microkernel family [34]. The L4 microkernel family is known for the formal verification ability and seL4 are therefore formally verified [39]. seL4 also functions as a hypervisor. seL4's capability scheme seems to focus more on capability in the execution environment and not so much about file storage. Both security in the execution environment and file storage are very important, but it has not been clear in the literature how exactly seL4 handles file storage, but as seL4 is a microkernel, the services lives in user mode and we assume that any file system could be attached, as long a service is implemented for it.

In the seL4 whitepaper [33] a classic example is given about how the capability should work. Assume Alice wants to compile a file. In this case Alice needs to hold three capabilities: an execute capability to the compiler, a read capability to the input file, and a write capability to the output file, this is exactly how a capability system should work. This solves the confused deputy problem as we introduced in Section 2.2 which cannot be solved with ACLs, which are also underlined in the seL4 whitepaper. In the paper they also state that if someone is trying to sell you a "secure" OS, it should not only have a correctness proof, but it should also be using capability-based access control. If not, the OS is not secure.

So in essence we need a capability system, which can be proven correct. The seL4 kernel can be formally verified from the source code itself, whereas ACRYLICS will focus on the correctness of the capability- and disk encryption-scheme. Furthermore the goal of

ACRYLICS is to combine the capability security in the kernel and the file system, such that it cannot be attacked, unless the whole protocol can be broken.

4.3 Remarks on File Systems

With the analysis so far, we have come across a lot of different security mechanisms. Another place developers are trying to enforce security is in file systems. Some of the features mentioned earlier in the chapter require special File System Attributes, like SELinux and Linux capabilities.

If we look at the classical NFS file system, it is simply not designed to hold such information and first in 2017, RFC8276 was proposed to add extended attributes to NFSv4. According to Microsoft classical FAT file systems did not have extended attributes [48], however, it seems that there exist solutions which uses hidden files to support it, but the point here being, that one should be aware of what the file system supports, and what is needed for the security wanted.

One thing is the support of extended attributes, another thing is encryption on file system level. Some Full Disk Encryption tools have already been introduced earlier in the chapter; BitLocker, FileVault and dm-crypt, but some file systems themselves have encryption. We will take a closer look at CryFS and ZFS.

4.3.1 CryFS

CryFS is a file system designed such that a user securely can store data at a cloud provider [45, 46]. CryFS encrypts the file contents, but also metadata and directory structure, this means that a cloud provider would have no idea of the data saved in the cloud.

CryFS is developed in a master thesis [45] and is proved secure. CryFS has been inspirational due the ACRYLICS project development, even though CryFS security model and proof technique is based on Kristian Gjøsteen security game-based approach [25], where ACRYLICS is based on the model of and Dupont [21] and the UC framework prove technique, which was described in Section 3.2.

CryFS is designed to be a one-user-system between host and cloud, whereas ACRYLICS has constraints to support multiple users at kernel environment level.

4.3.2 ZFS

ZFS is also an interesting file system since they are integrating encryption into the file system. There are not many file systems that offer native encryption and this is why ZFS is interesting, also given ZFS's support for very large disk space.

The OpenZFS Developer Summit 2016 talk ”ZFS-Native Encryption” presents the encryption in ZFS and contributes to awareness of procedures and key handling in such systems. Combined with the knowledge of Damgaard and Dupont [21] this made the concrete ideas for the implementation of ACRYLICS.

ZFS has also led to discussions about free space allocation, which has not been a main factor in ACRYLICS so far, but it needs serious considerations, since the performance of the allocation scheme can have huge impact on the overall performance, which are also seen in context to ZFS.

4.4 Discussion

In this section the findings from the analysis are discussed to define security levels 1 and 2 as shown in Figure 1.1. Based on the findings it is motivated why it is believed that the fundamental basis design for current Operating Systems can be improved towards better security.

4.4.1 Security Level 1 - No Disk Encryption

The first security level should be considered as a standard installation of an operating system without disk encryption.

A well known structure in operating systems in the separation of User Space and Kernel Space. There might be some more advanced distinction in the system, e.g. if we speak in the terms of which CPU ring level is used or if the separation is more in software.

Security Level
Level 4: ACRYLICS With improved capability module
Level 3: ACRYLICS
Level 2: Full Disk Encryption
Level 1: No Disk Encryption

However, conceptually it is about separating user operations from kernel operations. User operations are defined to happen in a user module (UM), which will be in the conceptual user space. In kernel space we will define multiple things. In security level 1 we need to define File Module (FM) and Disk Module (DM). FM handles the file system and the file operations to DM. DM is the actual disk and handles reading and writing to the physical disk. It is the kernel which handles the file system and disk IO, and therefore these two modules are placed in conceptual kernel space. Security level 1 aims at easily accessible security flaws, which often happens between what the running system protects and what the offline system protects and UM, FM and DM is used to illustrate this. The illustration of the conceptual system for security level 1 is shown in Figure 4.10.

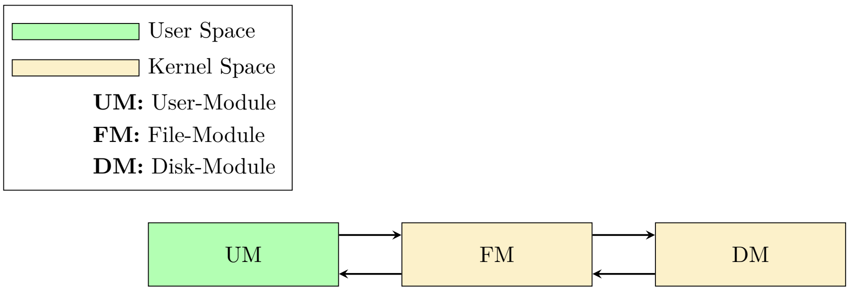


Figure 4.10: Standard System setup without Disk Encryption

To verify our conceptual understanding of the system we should take a closer look at a couple of attacks against this system. We should consider two types of attacks, namely a malicious UM, which would mean privileged escalation on a running system, and an offline attack against the physically stored data. The conceptual attacks are illustrated in Figure 4.11

It should be quite obvious that if the disk is not encrypted, the data on the disk can easily be read and the security of the system is broken. All non-encrypted systems are prone to this attack. In the analysis we saw that only Apple's new T2-chip products are encrypting something per default. So unless any configuration is made accordingly, any other system will be prone to this by default.

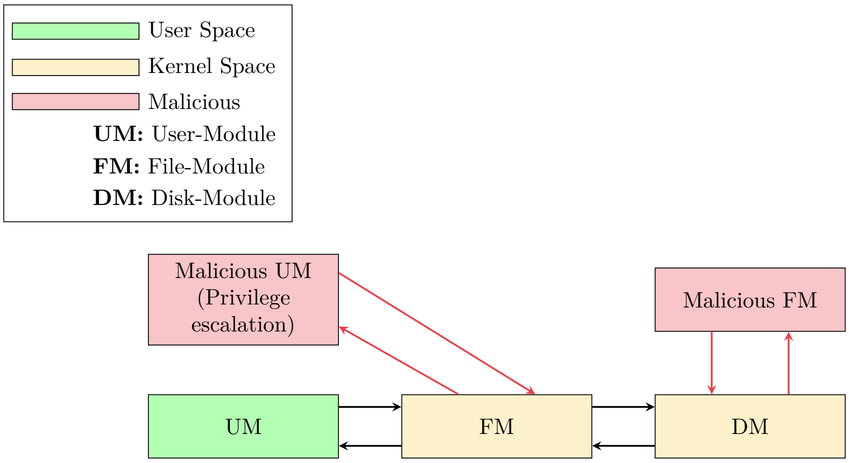


Figure 4.11: Standard System setup without Disk Encryption infected

Privileged escalation in the easy case could be obtained by booting a live USB system and inspecting the non-encrypted disk. The attacker will have root permissions on the running system and can access all data stored on the disk. This attack of course requires that the attacking system has a File Module that can understand the stored data, but for example a Linux system can read almost all file systems[43].

Privilege escalation attacks are most often more sophisticated, recall Jailbreak and attacks against android from subsections 4.2.2 and 4.2.4. Attacks against encrypted hard drives or privilege escalation through remote code execution [10].

As they also discuss in [43], disk encryption will prevent these attacks. With disk encryption we will add an extra module, the EM (Encryption Module). This is illustrated in Figure 4.12.

4.4.2 Security Level 2 - Full Disk Encryption

In Security Level 2 we will use the same conceptual model from security level 1, but we will add an Encryption Module in kernel space and will consider systems with Full Disk Encryption. Systems in security level 1 do not protect the data, and the security mechanisms analyzed does not prevent this and the next step is to add disk encryption.

The Encryption Module handles the encryption from FM to DM, with the same purpose as in the disk encryption scheme in Section 3.2. The updated model is illustrated in Figure 4.12

Security Level
Level 4: ACRYLICS With improved capability module
Level 3: ACRYLICS
Level 2: Full Disk Encryption
Level 1: No Disk Encryption

With added security of disk encryption, new attacks have to be invented. If the system should be violated from the perspective of a malicious UM, we should consider the case where the operating system itself holds the encryption key when the system boots and decrypts the disk content as needed for the user.

If somehow the system can be accessed with another user and the disk encryption is transparent, then it would be possible to read and modify the system since the operating system holds the key at all time [29, 28].

To breach the system physically by a malicious FM, the encryption key is needed since

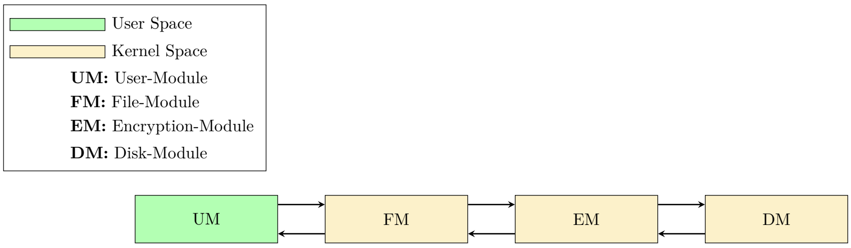


Figure 4.12: Standard System setup with Disk Encryption

the disk is encrypted. The attacker has to play the role of the malicious FM and malicious EM. There are multiple scenarios where it is possible to get the encryption key to the disk encryption solutions [40, 30, 11, 32].

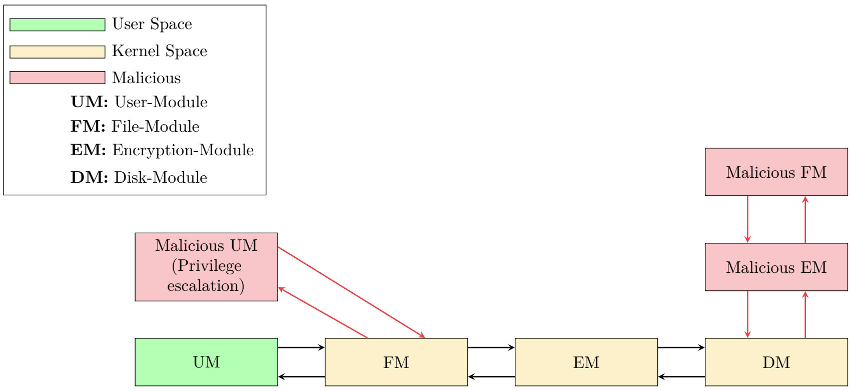


Figure 4.13: Standard System setup with Disk Encryption infected

Bypassing Local Windows Authentication to Defeat Full Disk Encryption

From Ian Haken's 2015 BlackHat talk ”Bypassing Local Windows Authentication to Defeat Full Disk Encryption”, he showed it was possible to bypass local authentication to bypass the full disk encryption on Windows. This is an example of a malicious UM attack.

HAFNIUM

HAFNIUM [28] is a series of vulnerabilities for Exchange Servers reported 2. March 2021 (CVE-2021-26855), which allowed unauthenticated Remote Code Execution with ”SYSTEM”-user permissions. It is possible for an attacker to instantiate a reverse shell, install additional malware, exfiltrate Exchange data or other things. Parts of the exploits were reported by the danish IT security company Dubex.

In a capability based system, such a service would not have capabilities to install other programs, such as malware and would prevent further escalation in such a breach. This attack is also an example of a malicious UM attack.

Cold Boot Attack

In a Cold Boot Attack an physical attacker makes a hard reset of a computer to dump the data in the RAM. Even though RAM is said to lose its data when it loses power, it actually remains for some time. Up to many minutes, and if the RAM is physically frozen it can remain in days and weeks.

This allows the attacker to read out the data in the RAM, most often the attacker will be looking for encryption keys to a potentially encrypted disk. If the attacker gets the encryption key, the attacker can play the role of malicious FM and malicious EM which are required to break the system on the physical storage [30].

LUKS - key management

It has been shown that it is possible to reduce the number of possible encryption to LUKS significantly [11] and this can lead to a breach of the encryption key. Hence an attacker can play the role of malicious FM and malicious EM to break the system on the physical storage.

It is no secret that key generation and derivation is hard and it is a tough task to do this right but only a single master encryption key is used in LUKS, so if this key is breached, then the whole disk can be decrypted.

VM FDE bypass

A very brutal attack is bypassing Full Disk Encryption (FDE) on Virtual Machines (VM). A virtual machine is running in a completely hostile environment since both the storage and the running system runs on a host. If the host is malicious it is very hard to defend against breach. In [32] it is shown how this can be done for Windows and Linux systems, with the FDE solutions dm-crypt, BitLocker and VeraCrypt. This is indeed an example of a malicious UM attack and the attacker will not even know the encryption key, but will still be able to breach the FDE.

Because of the transparent encryption that is used in the mentioned solutions, it is possible for the attacker to read anywhere on the disk the attacker wants.

Root Access

In common for Windows and the UNIX like systems is that a root or administrator user has access to anything. In the analysis different approaches were made to restrict this, but the conclusion is still that root can do anything.

This means that privileged escalation which results in root permissions on the systems will be able to do anything and only a real capability based system will be able to solve this, recall the confused deputy problem from Section 2.2.

Moreover it also means that administrators on the current systems always will have access to users files. It can both be a security problem but also a privacy problem. In very strict systems the administrators should maybe be allowed to access all files, but this should be a configuration and not the default, the users should be protected. Especially in the case where an attacker has root permission and current systems can access all data for all users.

5 ACRYLICS - Advanced Cryptografic Capability-based System

In this chapter ACRYLICS is designed and discussed. In Section 5.1 the core design is introduced. In sections 5.2 and 5.3, we will go into details about the capability scheme and the ACRYLIC File system respectively. The capability scheme is designed with theory from Section 2.3 and the file system is designed with theory from Section 3.2. In Section 5.4 the Encryption Module is explained in detail and how it works in ACRYLICS. In Section 5.5 the encryption flow is explained and it is pointed out that an improvement is needed to the capability module. In Section 5.6 the improvement is proposed and we will discuss how this solves the problem that we saw. Lastly in Section 5.7 **Security Level 3** and **Security Level 4** are defined and will be discussed with respect to the ACRYLICS design.

5.1 Core Design

ACRYLICS core design consists of five modules. The Capability Module which is built from the theory from Section 2.3 and the User Module, File Module, Encryption Module and Disk Module which are built from the four players, that was described in Section 3.2.

The definition of the following modules is the ACRYLICS interpretation of all the theory and analysis discussed so far.

5.1.1 User Module (UM)

In Damgaard and Dupont [21] users can call *Initialize*, *Boot*, *read* and *write*, this is also the case here. But in the ACRYLICS model, UM should be seen as the complete user interface as also defined in the security levels 1 and 2 in Section 4.4. This means that a user can also execute programs and the programs itselfs can also run and make requests as a ”player” in UM.

On *Initialize* and *Boot* UM sends the boot-password to EM, and EM is handling the request and sends true or false back. *read* and *write* requests from UM are sent to FM and corresponding capabilities are sent to CAPM for validation. If a capability is validated on a request, FM continues the request, otherwise access denied is returned.

For other requests such as executing programs or network access etc, UM would send requests to the specific service modules, for example a program dispatcher or Network Module, but on any action UM should send a correspondent capability to CAPM, such that CAPM can validate if the user or program has the necessary permission and the service module can either continue the process if access granted or return access denied if not granted.

5.1.2 Capability Module (CAPM)

The capability module is the access control mechanism and is based on the extended capability scheme introduced in Section 2.3. The capability scheme consists of three main structures which are Capability Objects, Capability Subjects and Capabilities. In Section 5.2 these structures are specified. Furthermore we will look into the sensitivity of the

Capability Objects, where it is possible to forge a more powerful capability or abuse the revocation mechanism.

The challenges are a major concern for us, because we want to bind the security of the physical storage and the system running. Often systems are assumed to always run and with kernel memory sensitive information can be hidden. This is not possible when saving all information offline, and a solution is needed.

In the ACRYLICS model CAPM gets input from UM in terms of capabilities and FM in terms of capability objects, CAPM returns access granted or denied to FM depending if the capability has the right permission to the capability object.

5.1.3 File Module (FM)

The File module is controlling the file system, but is also responsible for some other important things. In ACRYLICS the file system is the ACRYLIC File System specified in Section 5.3, but in theory this could be replaced with other file systems which follow the same requirements about binding users together between the file system and operating system. This means that the file module also has to handle all actions about user creation, deletion and have control over users current capabilities to files on the file system. All this information should then safely be stored in the file system. The reasoning here is that it should not be possible to mount the file system, without user credentials for a specific user, and only that user's files are accessible. To the best of our knowledge none of the analyzed systems does this. The analyzed disk encryption features only have one single master encryption key, where ACRYLICS will have an independent key per user, with the possibility via the capabilities to share encrypted files with a shared encryption key per shared file basis.

The File System should of course also handle where data is located on the disk, as well as manage free sectors. FM, hence the file system, operates on logical sectors and EM translates logical to physical sectors. So the file system will only know the logical sectors, and will pass on the request to EM. If FM on a request gets access granted from CAPM and passes the request to EM, EM will handle the request and return either success/fail on a write, or the actual data on a read. If access is not granted, then FM will return fail to UM.

5.1.4 Encryption Module (EM)

EM is the only module who knows encryption keys in the system. EM has an internal state where all the encryption information is stored. EM has assigned a special part of the disk where EM stores a Hash-Key-Tree which contains encryption keys and hashes for integrity of the entire disk. The state is restored on boot, with the boot-password. EM uses primarily a symmetric encryption scheme, with key material from the Hash-Key-Tree. If a specific user's data is encrypted, EM's key is XORed with the symmetric key derived from the user's password, hence the user's encryption key. Users can themselves choose another encryption scheme than encryption the scheme used by EM. The combination of different schemes and key sizes is implementation specific. The root of the hash-key-tree is encrypted with an asymmetric key pair, like the tree discussed in Section 3.2. EM is explained in detail in Section 5.4.

EM takes requests from FM and encrypt or decrypt the data depending on a write/read command. EM then either sends encrypted data to the DM for storage, or gets encrypted data from the DM, decrypts it, and sends the decrypted data to FM. EM translates LSI (Logical Sector indices) to LBA (the physical sector indices).

5.1.5 Disk Module (DM)

DM has access to the physical sectors on the disk. DM gets requests from EM and returns or writes data to requested physical sectors. DM is only working on encrypted data from the Encryption Module, so the Disk Module has no knowledge about the data.

The reason why this is called the disk module rather than only the disk, is that we see it from an operating system perspective, so it could be that there should happen some architecture specific thing in here, and then interfaces with the physical disk.

5.1.6 Complete Module Structure and Communication Flow

The modules are now introduced and in Figure 5.1 the complete structure is illustrated. If ACRYLICS' design is compared to the security levels 1 and 2, the CAPM module is the difference. CAPM is the central and enforced access control mechanism. Because there is an enforced capability based access control mechanism as part of the core design, this design stands out compared to the other analyzed systems. It is a change in the core design, and ACRYLICS cannot be adopted in the current systems. In the design discussion in Section 5.7, the security levels 3 and 4 are defined, it will be argued with examples, how it is a fundamental change in the core design of operating systems.

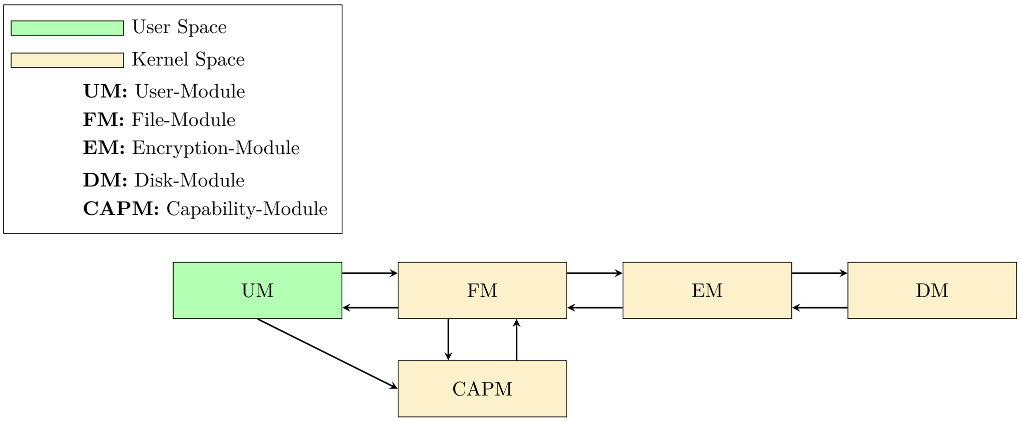


Figure 5.1: Complete module structure of ACRYLICS

To get more confident with the communication flow, a concrete example of a read request is illustrated in Figure 5.2.

Following the numbers on the arrows, a read FILE request is made to FM. FM sends the Capability Object (CAP-OBJ) which is stored with the FILE to CAPM. At the Same time the user sends the correspondent capability (CAP) to CAPM. CAPM then validates if CAP grants access to CAP-OBJ, and returns granted or denied back to FM. Based on the return value FM finishes the request with access denied or continues the process of reading the data and will at the end return the read data (decrypted by EM) to UM.

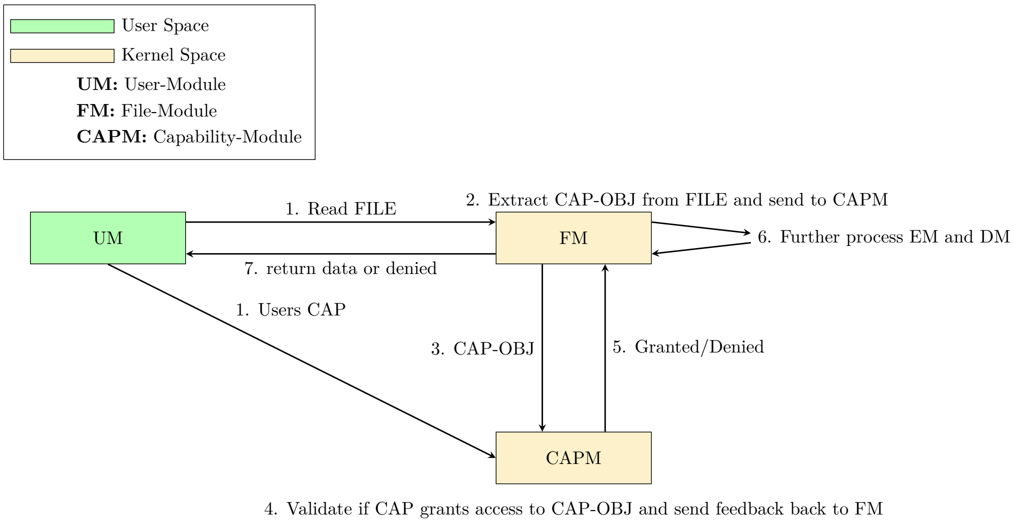


Figure 5.2: Read request to a file

5.2 Capability Module in Detail

In this section the structures of ACRYLICS' capability system are introduced, which are the functionality in CAPM. The capability system is based on Lopriore's [42] extended password capabilities as introduced in Section 2.3. The password derivation-, password validation- and capability reduction-operations in CAPM follows the pseudo code examples in Section 2.3.

5.2.1 Capability Subject

A subject is something which holds the capabilities itself, this could be a user on the system or a program etc. Subjects are also the ones who create capability objects. If a subject has created a capability object, this would be known as the owner of that capability object.

5.2.2 Capability Object

Everything in the system which requires access rights is considered as an **CAP-OBJ** (capability object). This could for example be files, resources or actions, e.g. all files are capability objects.

Every CAP-OBJ has a type, which for example could be a file. For a given object type some permissions are defined, for files this could be READ, WRITE and EXECUTE permissions.

In ACRYLICS CAP-OBJ is defined as follows in Figure 5.3

5.2.3 Capability

For every capability object created the creator gets an **owner capability** back. The owner capability is a capability with full permission to the given object. A capability can be reduced, where a new capability is created with a reduced set of permission. The reduced capability can be given to another subject and the subject will have the access rights specified by the reduced capability. A user which has got a reduced capability is denoted as a **shared user**.

Field names	Value	Comment
Object ID	Random 64-bit id	Used by the operating system to correlate capabilities and CAP-OBJ's.
Object Type	Enum value	Every object type defines permissions for objects of that type. Follows the theory in Section 2.3
Password	Random 64-bit	The secret for CAP-OBJ, this ”password” is used to construct and validate capabilities supplied to the CAP-OBJ. Follows the theory in Section 2.3.
Revocation Table	Binary mask with size of a reduction sub-field	Used to revoke capabilities of a class. The number of entries in the table can be set to some arbitrary number. Follows the theory in Section 2.3

Figure 5.3: Definition of **CAP-OBJ** (Capability Object)

In ACRYLICS a capability is defined as follows in Figure 5.4. Note that the Capability ID is the same value as the correspondent object's Object ID. The system stores the subject's capabilities in a hash table with capability ID as key. When access to a CAP-OBJ is wanted, the system can look up in the hash table with the Object ID and find the capability needed.

Field names	Value	Comment
Capability ID	64-bit id	The value is the same as the correspondent object's Object ID and is used by the operating system to correlate capabilities and CAP-OBJs
Calculated password	64-bit number	Generated on capability creation and is used when a capability has to be validated. Follows the algorithms introduced in Section 2.3
Reduction fields	sub-fields with bit size of how many permissions the correspondent CAP-OBJ has	Is the entry index in the revocation table in CAP-OBJ. Follows the theory in Section 2.3
class	number	Is the entry index in the revocation table in CAP-OBJ. Follows the theory in Section 2.3

Figure 5.4: Definition of **CAP** (Capability)

5.2.4 Challenges in Non-Improved CAPM

The data in CAP-OBJ is very sensitive. The proposed ACRYLIC File System in Section 5.3 is only supporting the non-improved CAPM scheme and two challenges are seen in this state. The challenges comes from the requirement that capabilities must be stored safely offline on the physical disk, where there is no ”kernel space” to hide sensitive information. ACRYLICS with the two proposed improvements in Section 5.6 is denoted “ACRYLICS with improved CAPM”.

With the non-improved CAPM scheme any user can access any shared CAP-OBJ since these are only protected with EM’s encryption key (Follows from the encryption flow explained in Section 5.5). Any user in the system is assumed to be able to boot the system hence knowing the boot-password and can restore the state of EM and hence knows EM’s encryption key. Non-shared CAP-OBJs are also protected with a user’s password and are therefore secure against other users. In Subsection 5.2.5 the technical challenges about the sensitive CAP-OBJ data are examined and in Subsection 5.6.1 an improvement for the CAPM scheme is proposed. The improvement requires additional information added to the ACRYLICS File System to support the proposed improvement.

The improvement will defend against any user can access any shared CAP-OBJ, but it will still be possible for a shared user to access CAP-OBJs shared to that user and possibly forge a more powerful capability or steal the file. But it will require that a valid user turn out to be malicious and has physical access to the hard drive.

The other challenge is a user with a revoked capability can exploit the revocation mechanism. The challenge is examined in Subsection 5.2.6 and an improvement is proposed. This improvement will also require some additional data in the ACRYLIC File System to support the improved CAPM scheme.

5.2.5 Forging Powerful CAP

The outputs in this subsection is from a real test run of ACRYLICS. It is used to illustrate and describe why CAP-OBJ’s data is too sensitive for a shared user to know. Recall that a shared user, is a user which has got a reduced capability to some CAP-OBJ.

In Listing 5.1 the data content of a CAP-OBJ is outputted. It includes the CAP-OBJ’s password. Recall the CAP-OBJ’s password is used to derive reduced capabilities passwords and is also used to validate the reduced capabilities.

```
|-------------------|--------------------------------|
|Capability object: |                                |
|-------------------|--------------------------------|
|Object id:         | 8de7e81bf854c27c               |
|-------------------|--------------------------------|
|Password:          |46e3fbf2abbacd29ec4aff517369c667|
|-------------------|--------------------------------|
|Revocation table:  | Value (only not zero entries)  |
|-------------------|--------------------------------|
|Entry number:   0  | ff                             |
|-------------------|--------------------------------|
|Entry number:   1  | f1                             |
|-------------------|--------------------------------|
|Entry number:   5  | f1                             |
|-------------------|--------------------------------|
```

Listing 5.1: Output of CAP-OBJ

Listing 5.2 shows a reduced CAP created by the owner of CAP-OBJ (form Figure 5.1). The reduced capability grants read permission for the shared user to CAP-OBJ.

```
|-----------------|--------------------------------|
|E-Cap:           |                                |
|-----------------|--------------------------------|
|Capability id:   | 8de7e81bf854c27c               |
|-----------------|--------------------------------|
|Reduction field: |    ff    |     1    |     1    |
|-----------------|--------------------------------|
|Password:        |273e26947eb6e23e9a99bb7e707cc86b|
|-----------------|--------------------------------|
|Class:           |1                               |
|-----------------|--------------------------------|
```

Listing 5.2: Output of CAP

A shared user has the knowledge needed to see a shared CAP-OBJ's data in clear. This is the case because we assume that any user is able to boot the system, hence knows the boot-password and can restore the state of EM. In ACRYLICS with the non-improved CAPM, shared CAP-OBJs is protected with key material only influenced by EM and the boot-password. The shared user can therefore decrypt CAP-OBJ and access the data if the shared user has physical access to the disk.

Listing 5.3 shows the process that CAP's password is derived and validated.

```
Encrypt first sub-field's value from CAP with CAP-OBJ's password as key:
CAP sub-field value = 1
CAP-OBJ's password  = 46e3fbf2abbacd29ec4aff517369c667
Calculated password = 92956f701a61a0505fe1f40555da2eb4

Encrypt value of next sub-field with the current calculated password as key:
Next CAP Sub-field value    = 1
Current calculated password = 92956f701a61a0505fe1f40555da2eb4
New calculated password     = 273e26947eb6e23e9a99bb7e707cc86b

For validation compare current calculated password and CAP's password:
Calculated password: 273e26947eb6e23e9a99bb7e707cc86b
Password from CAP:   273e26947eb6e23e9a99bb7e707cc86b
Passwords match:     Access granted
```

Listing 5.3: Derive and validation of CAP on CAP-OBJ

In this case the shared user has access rights ”1” since all sub-fields ANDed together equals 1, this means that the shared user has READ permission. But since the shared user can see CAP-OBJ in clear, the user can copy the CAP-OBJ's password to its own CAP and change the values in the sub-fields to all ones, which would mean full access like the owner has. See Listing 5.4.

```
|-----------------|--------------------------------|
|E-Cap:           |                                |
|-----------------|--------------------------------|
|Capability id:   | 8de7e81bf854c27c               |
|-----------------|--------------------------------|
|Reduction field: |    ff    |    ff    |    ff    |
|-----------------|--------------------------------|
|Password:        |46e3fbf2abbacd29ec4aff517369c667|
|-----------------|--------------------------------|
|Class:           |0                               |
|-----------------|--------------------------------|
```

Listing 5.4: Forged CAP with knowledge of CAP-OBJ

Even worse, the shared user can change data directly in CAP-OBJ, so the shared user could just change the password completely or change things in the revocation table. For example, the access for the original owner could be destroyed by changing CAP-OBJ's password and the shared user could keep access by updating its CAP accordingly, which would mean that the shared user has ”stolen” the file.

It is not possible to prevent a valid shared user from exploiting this, since the user has capability to access CAP-OBJ. However, with the improvement in Subsection 5.6.1, it is ensured that only a valid shared user can access it, and with the addition improvement of the revocation scheme, it is guaranteed that a revoked shared user cannot still access CAP-OBJ's data in clear, hence securing against a revoked valid shared user.

5.2.6 Abuse of Revocation Mechanism

Normally if a user changes an entry in CAP-OBJ's revocation table to all zeros; all access rights for that entry are revoked. The entry is a mask which is ANDed with the CAP's subfields, and if the result is zero, then all access is denied. Lets say that CAP-OBJ's owner makes such a revocation. Remember the owner does not have access to a shared user's capabilities, hence the owner cannot physically remove the capability from the shared user, i.e the shared user still knows where CAP-OBJ is and can still decrypt this because it still has enough knowledge to see CAP-OBJ in clear. The user can then either change the revocation table back or find another suitable entry in the revocation table and change its CAP accordingly.

Even if we assume that a shared user cannot access the CAP-OBJ in clear, nothing prevents the shared user to modify its own CAP and guessing another entry in the revocation table that might give the access rights that the user had.

In Subsection 5.6.2 the proposed improvement is preventing this completely.

5.3 ACRYLIC File System

In this section the core structures of the ACRYLIC File System is introduced. The requirement for the file system is to couple users of the file system and operating system together, such that it is not possible to use the file system without user credentials, hence binding the security of the offline system with the security of the running system. To do this it is necessary that the file system directly holds the information about the users, and not the operating system. The file system should also hold needed information to interface with CAPM and the capability scheme.

When the system is booted a specific process is followed in order to restore the state of EM and being able to decrypt relevant disk content. This is referred to as the encryption flow and is described in further detail in Section 5.5. The most important values for the encryption flow process are pointed out for each structure. Three dots in a figure means that the implementation has more values in the structure, but that these values are implementation specific.

The file system works with **LSI** (Logical Sector Index). **LBA** (Logical Block Address) refers to the real physical sector number. This means that FM uses LSI, and DM uses LBA. EM knows how to translate LSI to LBA and is explained in Section 5.4.

The point of LSI is that FM doesn't have to care about where the data is actually stored physically on the disk. FM can always assume LSI 0 as the first sector, up till LSI n, where n is the last LSI given the size of the partition.

The first structure is **MFSCI** (Master File System Control Information).
MFSCI is the only structure which has a fixed location on the partition. It is always placed at LSI 0 such that we always know where to find the information and pointers further into the file system that we need.

Field names	value
master_user_table	Logical sector index (pointer) to MUT
...	...

Figure 5.5: **MFSCI** (Master File System Control Information)

The most relevant value in MFSCI, is the pointer to **MUT** which holds the records of all users in the file system. It is important to note users are explicitly a part of the file system and these are the same users used by the operating system, hence there is no security gap between kernel and file system.

MUT (Master User Table) is a table with **MUTE** (Master User Table Entry) as entries. In the reference implementation the table is fixed to a number of 32 users, but this is implementation specific and could be dynamic, only bounded by the size of the disk to store user information.

Field names	value	size (entries)
entry	Master User Table Entry	32

Figure 5.6: **MUT** (Master User Table)

MUTE holds login-information about users. When a user logs in, the hash of the entered credentials is computed and compared to the stored hash in MUTE.

Field names	value
username	64 chars (bytes)
password (hashed)	128 bytes
user_master_information_table	Logical Sector index (pointer) to UMIT
...	...

Figure 5.7: **MUTE** (Master User Table Entry)

Furthermore MUTE has the pointer to the user specific information **UMIT** (User Master Information Table), which is encrypted under a symmetric encryption key derived from the user's password XORed with EM's encryption key.

All needed information for a user, is found in the users UMIT. The pointers to **UCT** and **RDT**, and the user's specific Encryption Settings are the most important values in the UMIT. Users can have different specific encryption settings, these will only be used for the user's own files.

UCT (User Capability Table) stores at least all file-capabilities for a user and is an arbitrary long list of CAP's. Important to note; capabilities for operations in the operating system kernel (syscalls) does not necessarily have to be stored here. The operating system can easily store these in files created and managed by the operating system. However capabilities for files itself are required by the file system to handle file access, and are therefore stored as a part of the file system.

Every user has its own **RDT** (Root Directory Table). This is a **DT** (Directory Table) but is the root of the file hierarchy and is therefore special. The LSI for RDT is found in the

Field names	value
master_user_table	Logical Sector Index (pointer) to RDT
capability_table	Logical Sector Index (pointer) to UCT
encryption_type	Enum value
encryption_algorithm	Enum value
encryption_mode_of_operation	Enum value
encryption_key_size	Number
encryption_key	Saved in bytes
...	...

Figure 5.8: **UMIT** (User Master Information Table)

user's UMIT. From RDT a user can reach all its own data, inclusive files shared with the user.

The DT holds entries **DTE** (Directory Table Entry). In the reference implementation the number of entries are set to 16, but this can be improved to be arbitrary.

Field names	value	size
entries	Table of DTE's	16 entries
...	...	...

Figure 5.9: **DT** (Directory Table)

DTEs contain the name of an object, for example the name of a file or directory. The field `table_entry_object_index` holds the LSI for the specific object.

This means if an object is shared, then different users will have a different entry in their own DT, but the pointer to the object would be the same, hence the object pointed to is exactly the same for the parties, and their individual capability are therefor checked against the same object and can lead to different access rights.

Field names	value
object_name	256 chars (bytes)
entry_object_kind	enum
table_entry_object_index	Logical Sector Index (pointer) to A-OBJ
...	...

Figure 5.10: **DTE** (Directory Table Entry)

A-OBJ (ACRYLIC Objects) are the objects in the ACRYLIC File System. Any A-OBJ holds a CAP-OBJ in its structure and A-OBJ can be seen as a CAP-OBJ with some additional data, as illustrated in Figure 5.11.

Field names	value	size
capability_object_block	CAP-OBJ	size of CAP-OBJ
...	...	...

Figure 5.11: **A-OBJ** (ACRYLIC Object)

5.4 Encryption Module in Detail

In this section EM is presented in detail. In Subsection 5.4.1 the formatting process is explained and the hash-key-tree is introduced and it is shown how it is constructed. The space overhead of the hash-key-tree is also considered. In Subsection 5.4.2 the read and write process for EM and below is explained.

5.4.1 Formatting the Partition

When the partition is formatted, some initial setup is required. In Damgaard and Dupont [21] it is suggested to use a hash-tree to handle integrity for the entire disk. The parent node contains the hash values for its child nodes. So on every update of a child, a new hash is calculated and updated in the parent, and this process happens from a leaf to the root, where the leaf holds the hash of a data sector.

It is also suggested in Damgaard and Dupont [21] to use different encryption keys in a similar tree structure. The leaf is encrypted with some random chosen key, the parent will save this key and the parent is encrypted with another random chosen key as well. This process also happens from leaf to root. This means that one has to be able to decrypt the root node to know any information on how to proceed decrypting the disk. As stated in Damgaard and Dupont [21] this ensures that if an adversary gets a snapshot of EM's state, it can only decrypt the current content on the disk and not data written after the snapshot was taken. Although it is possible for the adversary to produce new content to the disk until the next reboot, because the adversary knows the current state of EM and can update the state properly. But when the system is rebooted the private key is needed to decrypt the root node, and the private key is not held in EM's state, hence the adversary does not know the private key, and cannot decrypt the root node after a reboot.

In both trees it is suggested that an asymmetric key pair is used to encrypt the root node with the private key and generate a MAC for the root node. The public key is held in EM's state, such that EM can continuously encrypt the updated root nodes. Recall the root nodes will be updated on every write to the file system. Once the root nodes are decrypted with the private key at boot, the private key is erased in memory, such that the running system does not know the private key. The chosen *boot-password* is used to derive a symmetric encryption key to encrypt and decrypt this asymmetric key pair and calculate the MAC.

The *boot-password* can easily be changed, since the asymmetric key pair can be re-encrypted with key derived from the new password.

Hash-Key-Tree

In ACRYLICS the two trees are combined to the **Hash-Key-Tree** with three purposes. The Hash-Key-Tree is a b-plus tree and will hold the HASH value of its child node, the KEY to decrypt its child and the IV for the symmetric block encryption.

In Damgaard and Dupont [21] they are handling the situation about IV's differently as the mode of operation is assumed to be CBC, and by that some tricks can be done so the IV itself is not stored, but ends up with an extra cipher block instead. The extra chiper blocks are collected together in some physical sectors. However ACRYLICS want the ability to choose any arbitrary mode of operations, and it is therefore important to save the IV itself. The IV fits very well in the Hash-Key-Tree's design. On a decryption all the data needed is together in the particular node.

Recall that LBA sectors are the physical sectors on the disk. In the report and the reference implementation the physical sector size is fixed to be 4096 bytes. EM translates LSI (logical sector index used by FM) to LBA which EM uses. LBA-sector size is often seen as 512 bytes, which is the older standard. The standard is moving towards a sector size of 4096 bytes. In the case the system has a disk with LBA-sector size of 512 bytes, DM can be implemented to see the incoming LBA as size of 4096 bytes and then read/write 4 times 512 bytes for every read/write.

LSI-sectors size is chosen to be 8192 bytes. LSI 0 will always start at the first even LBA sector after the tree.

In the reference implementation every node in the Hash-Key-Tree has 64 entries with information to 64 child nodes. The reasoning about this comes from the LBA-sector size and size of the entries. As discussed the size of a LBA-sector is 4096 bytes, as each entry in the nodes are 64-bytes (see Figure 5.12), there is room for 64 entries in one LBA-sector $\frac{4096bytes}{64bytes} = 64$ entries and hence every node in the Hash-Key-Tree is therefore exactly the same size as a LBA-sector.

Field names	Value	Comment
HASH	16 bytes	Hash or MAC of the child
KEY	32 bytes	Random key chosen on every encryption. Max key size of 256 bits.
IV	16 bytes	Random chosen IV up to 128-bit which is the size of an AES-block-size

Figure 5.12: Definition of node entries

At formatting time the size of the partition is known and the size of the Hash-Key-Tree can be calculated statically. Pointers to the children are not needed because their location also can be calculated statically. Figure 5.13 illustrates the Hash-Key-Tree. Note that the figure already contains some user data, on a freshly formatted system it would only contain MFSCI and MUT, but to use the system at least one user is required, which creates UMIT, RDT and UTC for that given user.

Additional Special Sectors

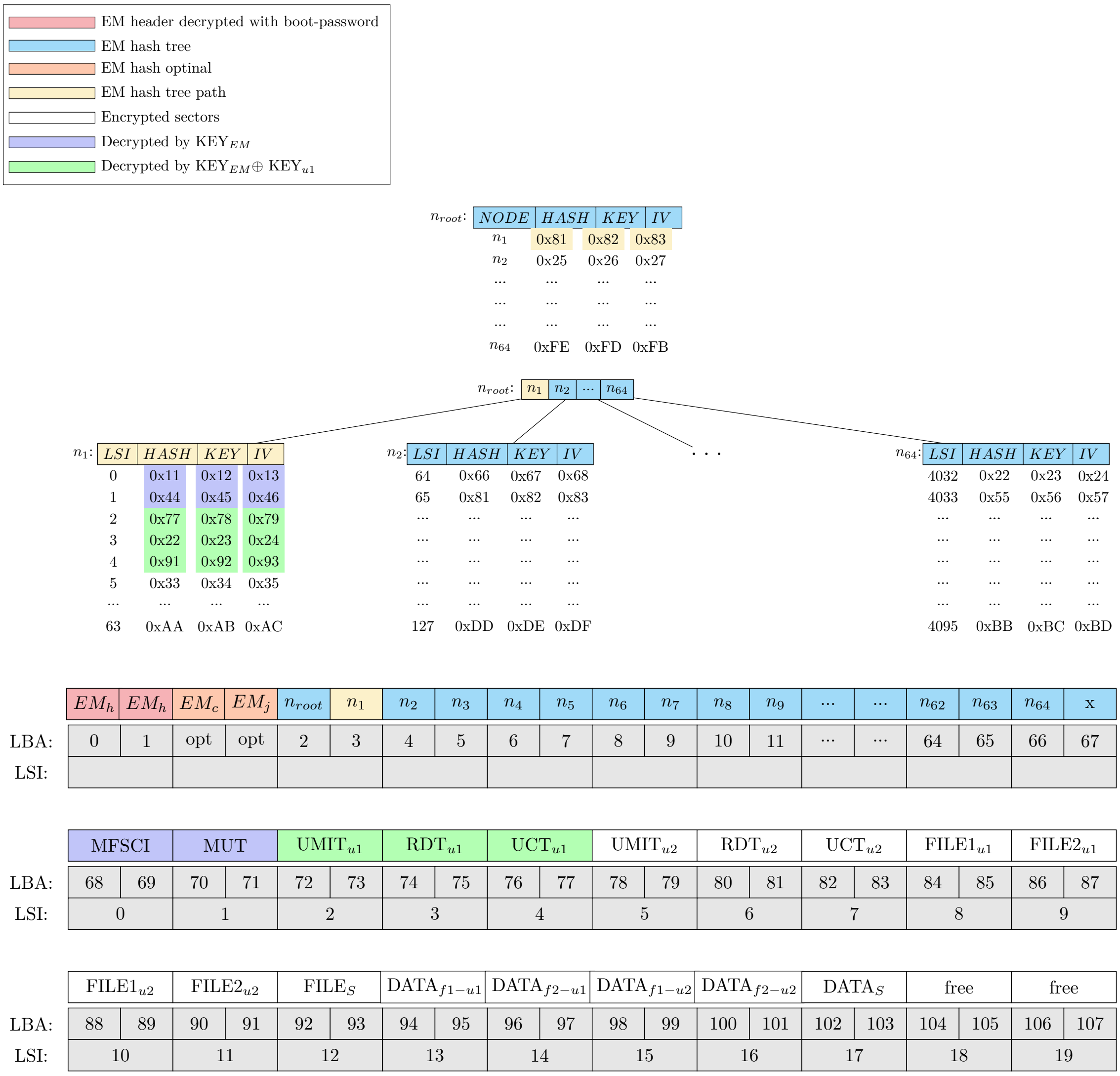
EM has some additional special sectors before the actual Hash-Key-Tree on the disk. In Figure 5.13 these are denoted EM_h, EM_c and EM_j. In Damgaard and Dupont [21] it is suggested that there can be a journaling and/or a Caching level below EM. EM_c (cache layer) and EM_j (journaling layer) are optional in ACRYLICS and is not designed, however the design considers room for implementing a solution at a later point. EM_h is EM header data. This include for example how many sectors there are use for EM_c and EM_j. In the reference implementation both of these are set to zero. EM_h also contain data about the LBA position on the disk, the size of the partition and the encryption settings for EM.

Most data in EM_h is encrypted with by the key derived from boot-password. But EM needs to know the encryption algorithm used for the encrypted data in EM_h to being able to decrypt. Therefore EM need some information about the encryption algorithm in unencrypted form, an therefore is the very first part of EM_h not encrypted. The non encrypted part only tells about the partition position on the disk and encryption settings, and does not reveal any sensitive data.

In EM_h's encrypted part the LBA offset to LSI 0 is stored. This is given to EM on every boot, so on a write request to a logical sector, EM knows how to calculate the LBA address that DM should write to, i.e. $LSI \cdot 2 + LBA\ offset$.

In Figure 5.13 the Hash-Key-Tree nodes is marked with blue, and the Hash-Key-Tree structure is shown in the top of the figure, and it is shown how the nodes are stored on the disk below. Marked with yellow is a path in the tree used in the example in Subsection 5.4.2.

n_{root} is the root of the Hash-Key-Tree. Recall n_{root} is encrypted with a public key pair. The public key pair and the MAC for n_{root} is also stored in the encrypted part of EM_h.



🟥 EM header decrypted with boot-password
🟦 EM hash tree
🟧 EM hash optinal
🟨 EM hash tree path
□ Encrypted sectors
🟪 Decrypted by KEY$_{EM}$
🟩 Decrypted by KEY$_{EM}\oplus$ KEY$_{u1}$

n_{root}:

NODE	*HASH*	*KEY*	*IV*
n_1	0x81	0x82	0x83
n_2	0x25	0x26	0x27
...	...	...	...
...	...	...	...
...	...	...	...
n_{64}	0xFE	0xFD	0xFB

n_{root}:

n_1	n_2	...	n_{64}

n_1:

LSI	*HASH*	*KEY*	*IV*
0	0x11	0x12	0x13
1	0x44	0x45	0x46
2	0x77	0x78	0x79
3	0x22	0x23	0x24
4	0x91	0x92	0x93
5	0x33	0x34	0x35
...	...	...	...
63	0xAA	0xAB	0xAC

n_2:

LSI	*HASH*	*KEY*	*IV*
64	0x66	0x67	0x68
65	0x81	0x82	0x83
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
127	0xDD	0xDE	0xDF

. . .

n_{64}:

LSI	*HASH*	*KEY*	*IV*
4032	0x22	0x23	0x24
4033	0x55	0x56	0x57
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
4095	0xBB	0xBC	0xBD

	EM_h	EM_h	EM_c	EM_j	n_{root}	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	...	...	n_{62}	n_{63}	n_{64}	x
LBA:	0	1	opt	opt	2	3	4	5	6	7	8	9	10	11	...	...	64	65	66	67
LSI:																				

	MFSCI		MUT		UMIT$_{u1}$		RDT$_{u1}$		UCT$_{u1}$		UMIT$_{u2}$		RDT$_{u2}$		UCT$_{u2}$		FILE1$_{u1}$		FILE2$_{u1}$	
LBA:	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87
LSI:	0		1		2		3		4		5		6		7		8		9	

	FILE1$_{u2}$		FILE2$_{u2}$		FILE$_S$		DATA$_{f1-u1}$		DATA$_{f2-u1}$		DATA$_{f1-u2}$		DATA$_{f2-u2}$		DATA$_S$		free		free	
LBA:	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107
LSI:	10		11		12		13		14		15		16		17		18		19	

Figure 5.13: Complete overview on Encryption Module

Space Overhead

Space overhead is a concern when introducing extra structures like the Hash-Key-Tree. By calculating the size of the tree and how much space the tree is pointing to, the space overhead can be calculated.

In the example in Figure 5.13 there is 65 nodes in the Hash-Key-Tree, this means $65 \cdot 4096$ bytes $= 266{,}240$ bytes is used for the Hash-Key-Tree. As 64 of the nodes each points to 64 leafs (logical sectors with data) the Hash-Key-Tree can reach $64 \cdot 64 = 4096$ logical sectors. Recall that the logical sectors is defined to be 8192 bytes, hence the Hash-Key-Tree can reach $4,096 \cdot 8,192 = 33{,}554{,}432$ bytes.

The overhead is then $\frac{266{,}240 bytes}{33{,}554{,}432 bytes} = 0.0079\%$.

The general formula for overhead can be calculated by the formula given in Equation 5.1 given the height h of the tree gives the overhead o. The overhead will always remain around 0.0079% with the sector sizes chosen here.

$$\frac{(\sum_{n=0}^{h} 64^{n}) \cdot 4096 bytes}{64^{h} \cdot 64 \cdot 8192 bytes} = o \tag{5.1}$$

The structural overhead in CryFS is calculated to 0.05% [45], so we should not be concerned about the Hash-Key-Tree overhead. Commonly the overall overhead of file systems are estimated to 3-5%. However it is difficult to say how the overhead should be calculated and which structures it should contain, so it would require a real investigation in file systems overhead to get concrete results.

However, the free sector allocation meta data is not implemented in the ACRYLIC File System yet and would definitely contribute to the final overhead of the entire ACRYLICS system, but the final overhead is not expected to be larger than the other systems.

5.4.2 Read and Write in EM and Below

With Figure 5.13 as the starting point, assume that user1 wants to read RDT$_{u1}$. FM issues a read request for LSI 3, where RDT$_{u1}$ is located. EM's job is to translate LSI to LBA, issue a read request to DM for the LBA and decrypt the data DM sends back.

In this case it would be $3 \cdot 2 + 66 = 72$ LBA, EM request DM for the data located at LBA 72. EM gets the data and will decrypt it. To get the KEY$_{EM}$ that was used to encrypt this block, EM has to traverse the tree. Starting from the root node, EM checks that LSI 3 is smaller than 64, and follows the first child node, the first child node is denoted n_1 and marked with yellow in Figure 5.13.

If n_1 is not cached into memory, EM has to ask DM to read and return the data in LBA 1, which is the LBA address for n_1. n_1 is ofcourse encrypted, but EM can get the key from the n_1's entry in n$_{root}$, marked with yellow in the figure. EM can validate the hash of n_1 with the hash from the entry in n$_{root}$ and use the KEY and IV to decrypt n_1.

Since n_1 is pointing to leaves, EM looks up the entry for LSI 3. EM can validate the hash form LSI 3, and can find KEY$_{EM}$ and IV which are used to decrypt LSI 3. Since LSI 3 is user1's data, EM XOR's KEY$_{EM}$ with KEY_u1 to get the effective encryption key. With the effective encryption key and the IV, EM can decrypt the data and send the decrypted data back to FM.

If we assume that user1 updates RDT$_{u1}$ and wants to write it back to the disk, EM gets a request to write RDT$_{u1}$ back to LSI 3. In this case EM will generate a new random key to be KEY$_{EM}$ and XOR this with KEY_u1. A new IV is also randomly generated and the data are encrypted with the effective key and the IV.

EM requests DM to write the encrypted data to LBA 72, calculated as before. EM has to update the tree and will write the updated KEY, IV and HASH into the LSI 3 entry in n_1, if the node is not cached, we have to decrypt the path in the tree as we did when we read the sector.

When n_1 has been updated the parent has to be updated, i.e. n$_r oot$. We generate a new key to encrypt n_1 and a new IV. Encrypt n_1 and calculate the new HASH, and update the entry for n_1 in n$_r oot$ with the new values.

As discussed earlier n$_{root}$ is encrypted by the public key of EM and a new MAC is calculated and stored.

5.5 ACRYLICS Encryption Flow Process

In this section the encryption flow is described in detail. Recall that MFSCI is the only structure with a fixed position, namely at LSI 0. MFSCI contains the LSI of the next structures needed in the boot process.

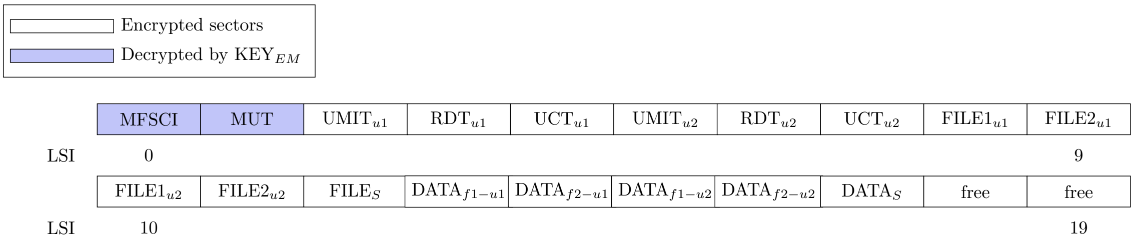
The same snapshot of the file system as in Figure 5.13 is used, this snapshot is used throughout the rest of this Chapter. The structures and data in the snapshot are placed in one arbitrary way. All other sectors than MFSCI could be located in any order.

The snapshot has two users, u1 and u2. Both users have two personal files, FILE1$_{u1}$, FILE2$_{u1}$ and FILE1$_{u2}$, FILE2$_{u2}$ respectively. The two users share a file, FILE$_S$. All files are A-OBJ's.

5.5.1 Step 1 - Booting with EM Key

Initially the system is not running and anything on the disk is encrypted, hence no plain text data can be derived. On system boot, the system will ask for the boot-password which derives the key to restore EM's state and EM's encryption key KEY$_{EM}$, recall Section 5.4. As discussed in Section 5.4, KEY$_{EM}$ is not just a single key, but is found via the Hash-Key-Tree. In the encryption flow model KEY$_{EM}$ refers to the correspondent key found in the leaf's entry for the correspondent Logical Sector.

KEY$_{EM}$ can decrypt MFSCI and MUT as illustrated in Figure 5.14.



⬜ Encrypted sectors

🟦 Decrypted by KEY$_{EM}$

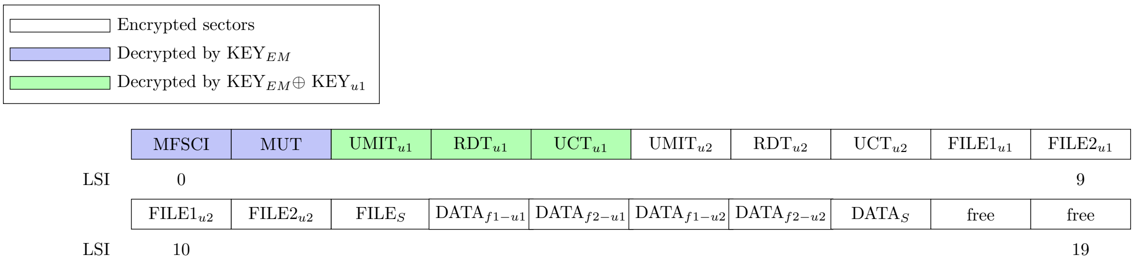
	MFSCI	MUT	UMIT$_{u1}$	RDT$_{u1}$	UCT$_{u1}$	UMIT$_{u2}$	RDT$_{u2}$	UCT$_{u2}$	FILE1$_{u1}$	FILE2$_{u1}$
LSI	0									9
	FILE1$_{u2}$	FILE2$_{u2}$	FILE$_S$	DATA$_{f1-u1}$	DATA$_{f2-u1}$	DATA$_{f1-u2}$	DATA$_{f2-u2}$	DATA$_S$	free	free
LSI	10									19

Figure 5.14: Step 1 - KEY$_{EM}$ decrypts MFSCI and MUT

5.5.2 Step 2 - Enter User Credentials for Active Session User

When MUT is loaded in step 1, the system will ask for credentials for a user to login. The entered credentials will be validated from the data in MUT. When a user is logged into the system and makes an action, the user is considered as the *active session user*.

Figures 5.15 and 5.16 show the state of decryption if the credentials is entered for u1 or u2 respectively.



⬜ Encrypted sectors

🟦 Decrypted by KEY$_{EM}$

🟩 Decrypted by KEY$_{EM}\oplus$ KEY$_{u1}$

	MFSCI	MUT	UMIT$_{u1}$	RDT$_{u1}$	UCT$_{u1}$	UMIT$_{u2}$	RDT$_{u2}$	UCT$_{u2}$	FILE1$_{u1}$	FILE2$_{u1}$
LSI	0									9
	FILE1$_{u2}$	FILE2$_{u2}$	FILE$_S$	DATA$_{f1-u1}$	DATA$_{f2-u1}$	DATA$_{f1-u2}$	DATA$_{f2-u2}$	DATA$_S$	free	free
LSI	10									19

Figure 5.15: Encryption state step 2 with u1 credentials

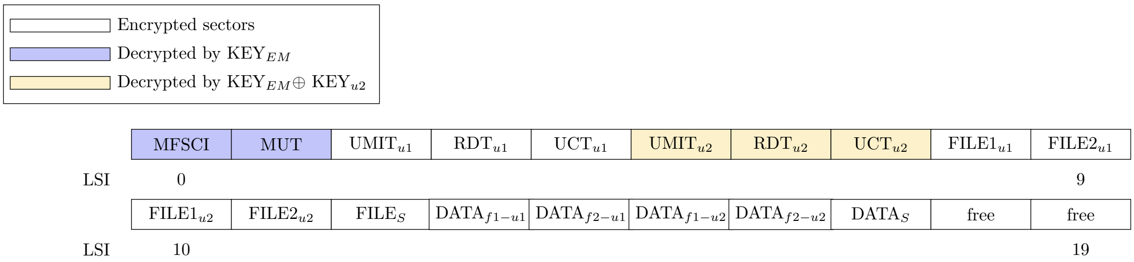


Figure 5.16: Encryption state step 2 with u2 credentials

When the *active session user* is logged in, it has access to its RDT. From the RDT the user can start reading files and directories which the user has access to.

Let's look at u1's RDT, see Figure 5.17, and note the LSI for FILE_S. Remember that FILE_S is a shared file between the two users. We assume that u1 is the owner, hence CAP_{fs-u1} is the owner capability for FILE_S.

File name	LSI to A-OBJ	Capability to access A-OBJ (Held by the Operating System Kernel)
FILE1	8	CAP_{f1-u1}
FILE2	9	CAP_{f2-u1}
FILE	12	CAP_{fs-u1}
...	...	...

Figure 5.17: RDT u1

If we look at RDT for u2 in Figure 5.18, the LSI pointer to FILE_S is also 12 and points to the exact same A-OBJ as in u1's RDT.

However u2 has its own reduced capability CAP_{fs-u2} which only provides reduced access rights to the A-OBJ.

File name	LSI to A-OBJ	Capability to access A-OBJ (Held by the Operating System Kernel)
FILE1	10	CAP_{f1-u2}
FILE2	11	CAP_{f2-u2}
FILE	12	CAP_{fs-u2}
...	...	...

Figure 5.18: RDT u2

5.5.3 Step 3 - Reading Data from Files

In this subsection the procedure for reading data from files is now described. We will first see an example of reading data from a non-shared file, and then an example of reading from a shared file.

Reading data from a non-shared file

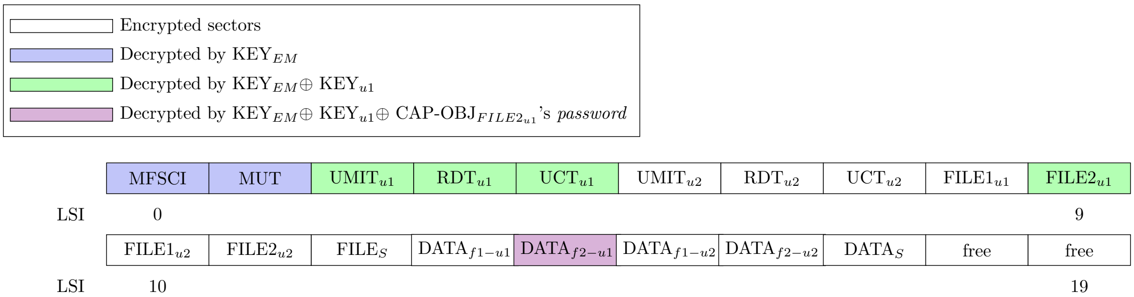
Let's imagine that u1 wants to read the data, DATA_{f2-u1}, from FILE2_{u1}.

u1 issues the read request to FM. FM finds the path for FILE2_{u1} in the directory tables. From the RDT in Figure 5.17 we see that FM needs to read $\text{A-OBJ}_{FILE2_{u1}}$ from LSI 9 and from the A-OBJ extract CAP-OBJ$_{FILE2_{u1}}$. A-OBJ$_{FILE2_{u1}}$ is encrypted by the user's encryption key, KEY$_{u1}$. EM have access to KEY$_{u1}$ since u1 is active in the system, so FS asks EM to get the content for the block containing A-OBJ$_{FILE2_{u1}}$. EM decrypts it with KEY$_{EM}\oplus$ KEY$_{u1}$.

From the decrypted A-OBJ$_{FILE2_{u1}}$ FM extracts CAP-OBJ$_{FILE2_{u1}}$ and proceed with the request by providing this to CAPM. Alongside u1 provides CAP$_{f2-u1}$ to CAPM (recall Figure 5.2). CAPM can now validate if u1 have the permissions to read the content of FILE2$_{u1}$.

In this example access is granted. FM passes the last part of the encryption key to EM. In the case where the file is non-shared, the correspondent CAP-OBJ's password is used as the last part of the encryption key. So in the example CAP$_{f2-u1}$'s password is the last part of the effective encryption key, hence the effective encryption key to decrypt DATA$_{f1-u2}$ is KEY$_{EM}\oplus$ KEY$_{u1}\oplus$CAP-OBJ$_{FILE2_{u1}}$'s password. This is illustrated in Figure 5.19.

EM decrypts DATA$_{f1-u2}$ with the effective key and return the plain data to FM and FM can complete the request.



⬜ Encrypted sectors
🟦 Decrypted by KEY$_{EM}$
🟩 Decrypted by KEY$_{EM}\oplus$ KEY$_{u1}$
🟪 Decrypted by KEY$_{EM}\oplus$ KEY$_{u1}\oplus$ CAP-OBJ$_{FILE2_{u1}}$'s *password*

	🟦 MFSCI	🟦 MUT	🟩 UMIT$_{u1}$	🟩 RDT$_{u1}$	🟩 UCT$_{u1}$	UMIT$_{u2}$	RDT$_{u2}$	UCT$_{u2}$	FILE1$_{u1}$	🟩 FILE2$_{u1}$
LSI	0									9
	FILE1$_{u2}$	FILE2$_{u2}$	FILE$_S$	DATA$_{f1-u1}$	🟪 DATA$_{f2-u1}$	DATA$_{f1-u2}$	DATA$_{f2-u2}$	DATA$_S$	free	free
LSI	10									19

Figure 5.19: Encryption state step 3 u1 reads FILE2$_{u1}$

Reading data from a shared file

When reading data from a shared file all subjects which have a capability to the object need to be able to decrypt it. In particular A-OBJ is the problem. A-OBJ is in the non-shared case encrypted with the user's password XORed with KEY$_{EM}$. A-OBJ is needed to extract CAP-OBJ such that FM can give this to CAPM for validation, and importantly the extracted CAP-OBJ's password is used to decrypt the data A-OBJ points to. So without the decrypted A-OBJ, it is not possible to decrypt the data at all.

Let's consider that u2 wants to use its *read only*-permission to read the shared FILE$_S$ owned by u1. The two users must agree in a way to decrypt A-OBJ$_{FILE_S}$. The non-shared approach would be to use either KEY$_{EM}\oplus$ KEY$_{u1}$ or KEY$_{EM}\oplus$ KEY$_{u2}$ to decrypt A-OBJ$_{FILE_S}$. But this cannot be the case since u1 does not, and must not know u2's key and vica versa. So the naive approach is to let EM use its key, KEY$_{EM}$, as an independent authority. Both users can now get access to A-OBJ$_{FILE_S}$ and can continue as the procedure in the non-shared case, where CAP-OBJ's password is used as part of the key to decrypt the data. So CAP-OBJ$_{FILE_S}$'s password$\oplus$ KEY$_{EM}$ gives the effective encryption key to decrypt DATA$_S$. This is illustrated in Figure 5.20

However, with this scheme we can observe that any user on the system which knows KEY$_{EM}$ will be able to decrypt the content of A-OBJ$_{FILE_S}$ and hence also be able to decrypt the data. Even users that do not have a capability for the file.

We can also observe that u2 has access to the decrypted A-OBJ and hence knows CAP-

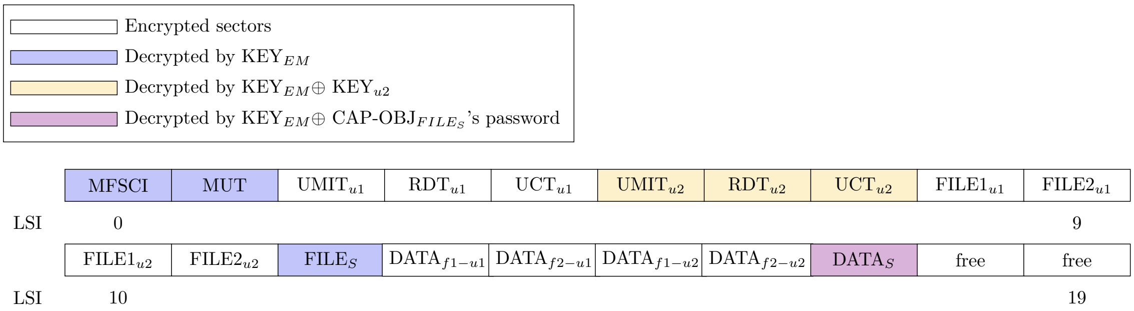


Figure 5.20: Encryption state step 3 u2 reads shared FILE

OBJ's password and can construct a more powerful capability or can steal the file from the original owner. Recall the discussion about sensitive CAP-OBJ data in Subsection 5.2.4.

5.6 Improvement of CAPM Scheme

In this section improvements are proposed to observations made last in Section 5.5. The issue about whether any user can access a shared CAP-OBJ can be solved, and the improvement is described in Subsection 5.6.1.

It is not possible to prevent a valid user from being malicious, however, it is possible to revoke a user's capability, such that a revoked user can no longer decrypt CAP-OBJ, and modification of CAP-OBJ is prevented. It is also prevented so that the user can modify its own CAP and guess another entry in the revocation table which could give the capabilities back. This improvement is described in Subsection 5.6.2.

As mentioned before, ACRYLICS with these two improvements integrated is denoted as "ACRYLICS with improved CAPM".

5.6.1 Shared Object - Shared Secret

At Figure 5.20 we saw the naive approach, where EM's keys were used to encrypt and decrypt the shared A-OBJ. Since we assume that KEY$_{EM}$ is known by all users, this cannot be considered safe. However we know from the analysis in Chapter 4, that most modern computers have a built-in security chip e.g. the **TPM** (Trusted Platform Module). We know the TPM is designed to hold and use encryption keys securely without anyone being able to know the keys.

Assume if we would make use of the TPM and let it manage EM's key. This would mean that only the TPM would be able to encrypt and decrypt shared A-OBJs from where the sensitive CAP-OBJs are derived. Since no user would be able to decrypt the data directly from the offline physical storage, this would infer some security and could solve the problem. However, there would also be consequences and concerns by doing this. The most concerning would be that the data is bound to the hardware and ACRYLICS would require systems to have a TPM. From the analysis in Chapter 4 some weaknesses about the use of the TPM chip are also raised and one could imagine that other attacks suddenly were possible.

For example if an attacker somehow could get access to the encryption key or with some extreme privileged escalation could ask the TPM for decryption. This would not only compromise the active user, but any shared content would be exposed for decryption, because it only relies on KEY$_{EM}$.

Introducing the Shared Secret

A shared secret is needed between sharing parties and it has to be bound to a CAP-OBJ. When a CAP-OBJ is created the owner will get a random generated secret and the owner capability back. The owner can then distribute it when generating and sharing reduced capabilities. Each shared user can encrypt the shared secret with their own encryption key, and the shared secret will be stored safe, and only the shared users know it.

The shared secret is combined with KEY$_{EM}$. $secret \oplus$KEY$_{EM}$ will be the effective encryption key for the shared A-OBJ which contains the sensitive CAP-OBJ.

The secret is added to the user's DTE from Figure 5.10, the modified structure is illustrated at Figure 5.21. The secret will have the size of the encryption key chosen for the encryption.

Field names	value	size
object_name	chars	256 bytes
entry_object_kind	enum	2 bytes
table_entry_object_index	LSI (pointer)	8 bytes
capability_object_secret	random number	size of encryption key
...	...	...

Figure 5.21: Modified Directory Table Entry structure

In Figure 5.22 the updated encryption flow using the shared secret is illustrated.

With the shared secret it is only possible to read and decrpyt A-OBJ$_{FILE_S}$, with a DTE containing the correct secret. An adversary only knowing KEY$_{EM}$ will no longer be able to decrypt the data, and the issue is solved. It now requires some specific knowledge to read a shared file.

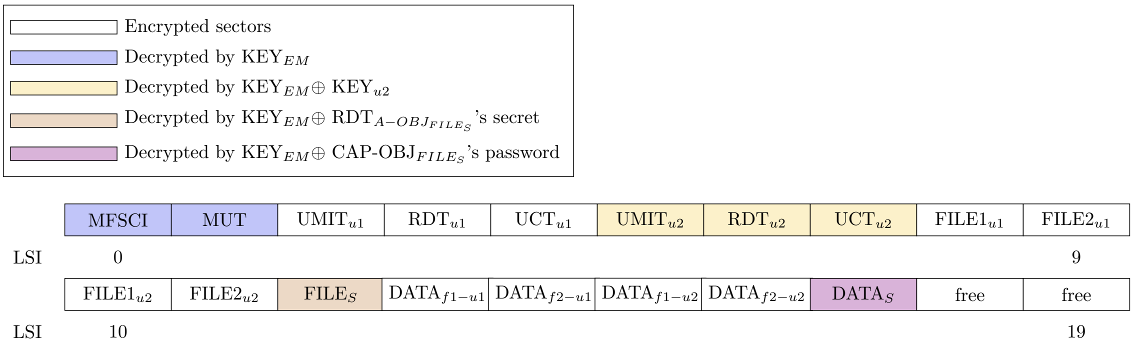


Figure 5.22: Updated encryption state step 3 u2 reads shared FILE$_S$

It could also be considered if CAPM could have a secret key (KEY$_{CAPM}$) stored in a hardware module like the TPM. Then all CAP-OBJ's and CAP's in the system could be encrypted with KEY$_{CAPM}$. If we look at the encryption state in Figure 5.20, this would actually improve the situation since CAP-OBJ$_{FILE_S}$ would be encrypted and an adversary would not be able to forge a CAP, because CAP-OBJ$_{FILE_S}$'s password is encrypted with the not known KEY$_{CAPM}$. However, adversaries would still have some access to A-OBJ's data which we don't want to allow. But with the proposed shared secret improvement, then the proposed improvement would prevent outsiders from accessing A-OBJ and KEY$_{CAPM}$ would prevent adversaries and malicious users from accessing the sensitive CAP-OBJ data.

In practice, this would again imply that a hardware module like the TPM is required, and as discussed before we don't want that dependency. But in an environment where extra security is wanted and TPM is available, this solution could be worthwhile to investigate further.

5.6.2 Add Seals to the Revocation Table

The shared secret improvement prevents users from accessing shared A-OBJs which are not allowed. But a shared user with a valid capability can still decrypt A-OBJ and get access to the sensitive CAP-OBJ. As mentioned before a malicious user cannot be prevented, but in this subsection an addition to the revocation scheme is proposed, to make sure that only users with an active capability (a capability which has not been revoked) can access the sensitive data.

The proposed improvement is an addition to the revocation table in CAP-OBJ. We want to encrypt CAP-OBJ inside A-OBJ, but want an unencrypted part of the revocation table containing seals, but still stored in A-OBJ. A new "encryption secret" is added and is the encryption key used to encrypt and decrypt CAP-OBJ. We denote the new encryption key as $EncKey$.

A seal is a bit string which is XORed with $EncKey$. A shared user will have a sealed version of $EncKey$, and only if the same seal is used to unseal, the real $EncKey$ is revealed. The idea to seal the encryption keys came with inspiration from Dan Mossop and Ronald Pose's [52] "Information Leakage and Capability Forgery in a Capability-Based Operating System Kernel". If the rules about the seals are designed carefully, it is a strong addition to the revocation scheme.

With the proposed seal improvement a user with a revoked CAP is prevented from accessing the sensitive CAP-OBJ, and can no longer forge capabilities or manipulate CAP-OBJ when revoked. The chance of guessing another entry (seal) that will work, would be the same probability as guessing $EncKey$ itself, which is with negligible probability. It is not possible to prevent the user from being able to decrypt A-OBJ, but the added $EncKey$ is also used to encrypt and decrypt the data A-OBJ is pointing to. So far CAP-OBJ's password has been used as a part of the data encryption key. However it is not possible to change a CAP-OBJ's password, as it would invalidate all reduced capabilities to the CAP-OBJ. A revoked user could have stored CAP-OBJ's password and would then be able to decrypt the data. However $EncKey$ can be changed, so if a class is revoked completely in the revocation table, $EncKey$ is changed, the data is re-encrypted with $\text{KEY}_{EM} \oplus EncKey$, and the revoked user can no longer decrypt the data.

The updated encryption flow with the seal improvement is illustrated in Figure 5.23

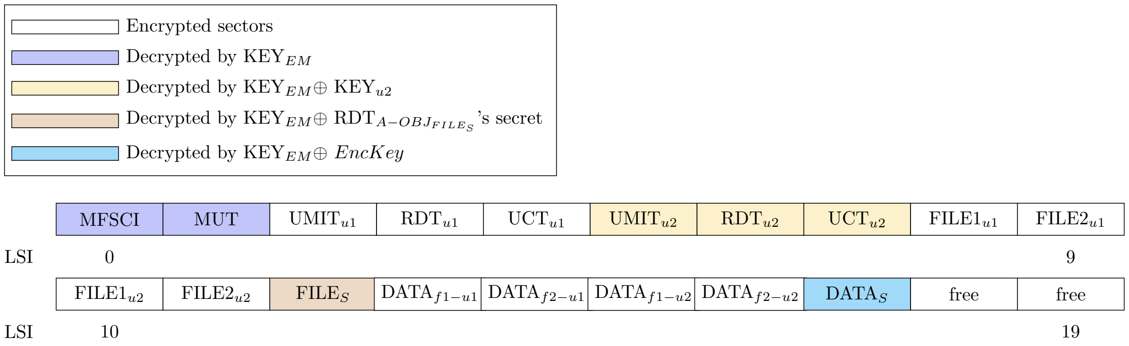


Figure 5.23: Updated encryption state step 3, with use of the seal improvement, u2 reads shared FILE$_S$

Extend DTE with Encryption Secret

The DTE from Subsection 5.6.1 are extended with the `cap_obj_encryption_secret` field, which is used to store *EncKey* or a sealed version of *EncKey*.

The newly added part of the revocation table is needed in clear text inside A-OBJ, since the seals are used to unseal the *EncKey* which is used to decrypt CAP-OBJ. Remember that A-OBJ is still encrypted as the shared secret improvement suggested.

When a new CAP-OBJ is generated a random encryption key are generated and given to the owner, and is stored in the user's DTE in the `cap_obj_encryption_secret` field. This is the real unsealed encryption key, *EncKey*. We assume that the owner will always be able to decrypt the object and is therefore seen as the authority and holds the real unsealed encryption key in its DTE. Shared users will have a sealed version of *EncKey* stored in the `cap_obj_encryption_secret` field.

Field names	value	size
object_name	chars	256 bytes
entry_object_kind	enum	2 bytes
table_entry_object_index	LSI (pointer)	8 bytes
capability_object_secret	random number	size of key size
capability_object_encryption_secret	random number	size of encryption key
...	...	...

Figure 5.24: Modified Directory Table Entry structure

Defining Sealing Rules

The sealing rules are now defined and the following terminology is used in the equations.

- $EncKey$ = The random generated unsealed encryption key used to encrypt and decrypt the sensitive CAP-OBJ.
- $Password$ = The password supplied from a user's CAP.
- $revocation[class]$ = The seal in the unencrypted part of the revocation table in CAP-OBJ, where *class* refers to the index in the revocation table. The value is in CAP's class field.
- $Secret$ = The additional proposed encryption secret in DTE. For the owner this is the *EncKey*, for other users this will be the *Secret* as is described in the following
- $MasterSecret$ = Password $\oplus$ Secret

EncKey derivation

The owner of an object has direct access to the *EncKey* stored in DTE as already discussed. Shared users has to derive *EncKey* as follows:

$$EncKey = Password \oplus Secret \oplus revocation[class] \tag{5.2}$$

Add a Class to Revocation Table

A new entry in the revocation table is created at the same moment as the first reduced capability for a given *class* is created. A random number for *Secret* is generated and is handed out to the user of the newly reduced capability. The creator of the reduced capability already knows *EncKey*. *Password* is derived as normal when creating a reduced capability. With this information the new seal can be calculated for index *class* in the revocation table as follows:

$$revocation[class] = Password \oplus Secret \oplus EncKey \tag{5.3}$$

Derive Encryption Secret From Existing Class

If the *class* decided for a reduced capability already exists, the *Secret* is derived as follows in Equation 5.4 and is given to the user. *Password* is derived as normal when creating a reduced capability. The creator of the reduced capability already know or can derive *EncKey* and can therefore calculate the secret as follows:

$$Secret = Password \oplus revocation[class] \oplus EncKey \tag{5.4}$$

Revoke a Class

On a class revocation, the bits in the revocation table at the index *class* are scrambled. A new random *EncKey* is generated and A-OBJ and the data pointed to by A-OBJ is re-encrypted. As *EncKey* is changed, all existing CAP's to other classes will result in a wrong derivation. So these have to be updated.

Password, *Secret* and *EncKey* are used to calculate the revocation entries. The *Password* and *Secret* are only known by a user itself and nobody else. We don't want anyone else to know this information, because that is sensitive information.

However with the proposed construction all entries in the revocation table can be fixed without knowing *Password* and *Secret* for all the other shared users.

In the following equations we see that the $MasterSecret$ can be derived for any class in the revocation table. The $MasterSecret$ is used to calculate the new seals for every entry in the revocation table, which makes sure that any shared user will drive the new *EncKey* correctly and retain the access.

$$MasterSecret = Password \oplus Secret \tag{5.5}$$
$$EncKey = Password \oplus Secret \oplus revocation[class] \tag{5.6}$$
$$EncKey = MasterSecret \oplus revocation[class] \tag{5.7}$$
$$MasterSecret = revocation[class]_{old} \oplus EncKey_{old} \tag{5.8}$$
$$revocation[class]_{new} = MasterSecret \oplus EncKey_{new} \tag{5.9}$$

From Equation 5.8 and Equation 5.9 we see that only the old *EncKey* and the old revocation table are needed to calculate a class's $MasterSecret$. This is exactly what a user with an active capability will know.

This means that a user can change *EncKey*, calculate the new revocation table and scramble *revocation*[*class*] for the *class* to revoke.

If a user supplies a CAP with a class which has been revoked, then the calculation for *EncKey* will result in some random bits, and CAP-OBJ cannot be decrypted properly. CAPM will reject the capability and the user will not be granted access and cannot decrypt the data pointed to by A-OBJ, since it also requires the correct *EncKey*.

This also prevents a user with a revoked capability to change anything in CAP-OBJ or to forge a new valid capability. Even though the revoked user can see the unencrypted revocation table, it is still not possible for the revoked user to calculate the new *EncKey*. This is true because one has to know *revocation*[*class*] and either *Secret* or *EncKey* to be able to calculate one or the other, and the user does not know either *Secret* or the new *EncKey*. The probability of guessing any of the values would be the same probability as just guessing the encryption key itself, which is negligible probability.

Final Suggestions

A user with a valid CAP to some CAP-OBJ will still know CAP-OBJ's password. The user is still able to forge a CAP with a full permission set or modify CAP-OBJ, as we saw in Subsection 5.2.4. But where a user before with a revoked capability could construct and/or modify CAP and CAP-OBJ, it is now **only** a user with an active capability and who knows the active *EncKey* which can do it.

If ACRYLICS should defend against a malicious user, some form of external authority is needed. It could be in the form of the TPM as discussed earlier with a secret KEY_{CAPM}, or a completely external CAPM could be used as we know it from certificate authorities and TLS handshake. In this way the capabilities would only appear encrypted for the users, and would only be decrypted at the authority where access would be validated, and access granted or denied would be returned.

The solution would most likely bring a lot of other challenges into play, but if security is needed in a larger organization, this could be the way to go.

Lastly, a revoked user still has some information about CAP-OBJ's password and would of course still know all the information which is in its own CAP. It also knows the shared secret to decrypt A-OBJ. Therefore it is suggested that the final solution should let *EncKey* encrypt the A-OBJ, such that a revoked user knows as little as possible. Of course the additional part of the revocation table still has to be unencrypted, but this should not lead to security concerns, since it will just appear as random bits. One should note that the encryption we purpose with CAP-OBJ and A-OBJ is happening above EM and the requirement for EM security is still fulfilled with the encrypt purposed in Section 5.6.1, this would guarantee security against outsiders, and then the encryption above EM would protect against a revoked user.

5.7 Design Discussion and Security Analysis

In this section the design choices are discussed and security levels 3 and 4 are defined and it will be shown why ACRYLICS is in another security level.

When security levels 1 and 2 were designed, the starting point was some attacks which successfully could be used against systems in these levels. First will be shown that ACRYLICS prevent these attacks and this defines Security Level 3, then we will come up with a specific attack against the ACRYLICS system, which makes room for improvement. In Security Level 4, the best security level, these attacks will be prevented, and we will discuss what types of attacks cannot be prevented.

5.7.1 Security Level 3 - ACRYLICS

It is easy to state that ACRYLICS are at least in Security Level 2, since the system have full disk encryption by default.

We left off Security Level 2 with privilege escalation attacks, and saw that transparent encryption is actually not as good as it sounds. We also saw that if someone can get the encryption key then all of the solution that we have analyzed is broken since they only use one single master key for the encryption.

Security Level
Level 4: ACRYLICS With improved capability module
Level 3: ACRYLICS
Level 2: Full Disk Encryption
Level 1: No Disk Encryption

Figure 5.25: Security Level 3

In Figure 5.26 attacks against ACRYLICS is illustrated. In the left side we see the UM privileged escalation box, but it is claimed to be prevented. why?

The ACRYLICS design uses capabilities and there is no overall root. So even though ACRYLICS often will have some ”System Root User” this will not lead to privileges where the entire system is broken. Files and data for all other users are safe, because it is encrypted with their own key.

It would probably still be the worst case if the ”System Root User” was compromised completely, as it would most likely have access to operating system files, but at least ACRYLICS secures other users data.

However we should consider how such a privileged escalation could appear. Recall the HAFNIUM attack introduced in Security Level 2, Subsection 4.4.2. We should not consider the attack concretely but the fact that some service is running and is exposed towards the network. If the service was run by another user than ”System Root User” in ACRYLICS, it would be a good case. If the service was compromised, it would never go worse than the service user was compromised. ”System Root User” and other users files would be safe. But even if the service was run by the ”System Root User”, it should be underlined that the service itself are given the capabilities needed to perform the service. Recall that programs are also Subjects in the capability scheme and hence also have to provide capabilities to CAPM.

If the service has been given capabilities according to principle of least privileged as introduced in Section 2.1. The compromise would only affect the smallest part of the system possible and hopefully this would not include the core system files.

If a worst case privileged escalation attack should be performed, the attack would need to steal the capabilities from the other users. But as a user’s capabilities are encrypted with

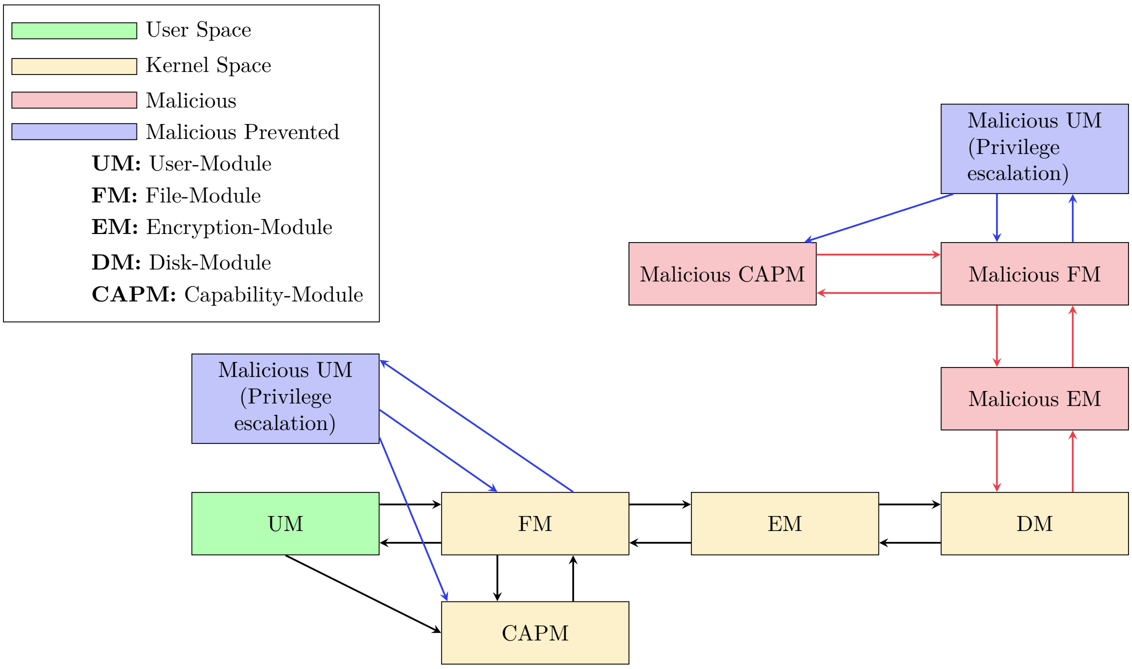


Figure 5.26: ACRYLICS

the user's key, it is almost impossible to steal capabilities from other users, and certainly from multiple users, since every user uses its own key to encrypt.

We should also consider the Cold Boot Attack introduced in Security Level 2, Subsection 4.4.2. This is piratically how to take a snap shot of EM. Since EM follows the scheme in Damgaard and Dupont [21], it can be guaranteed that the attacker will not learn more than the data currently on the disk. In addition only users which are logged into the system will be effected. If a user is not logged in, hence has not typed its password, there is no way to derive the user's specific encryption key. This will secure other user in the system on a breach.

In the right side of Figure 5.26 the offline storage attack are illustrated. In Security Level 2 it was possible to decrypt the system, if an attacker somehow could get the single master key used for the disk encryption.

Because ACRYLICS bind the security of the running system with the security of the offline storage, it is not possible to decrypt the disk, without "being" the user. An attacker will need both the boot-password and a user's password. Furthermore all the capabilities are needed to decrypt each file, so the attacker needs the capability list for the user to compromise. So to compromise one user an attacker will need the boot-password (or the key derived from the password), the user's password (or the key derived from the password) and the user's capability table. The attack basically has to know the same amount of information as the user, hence the attacker is "being" the user. If additional users should be attacked, the attacker needs the user password (or the key derived from the password) from each user.

As stated with privileged escalation there is no single super-user which can access it all, hence there is no privileged escalation which makes it possible to read the entire disk without "being" all the users in an offline attack.

Since attacks for both the running and the offline system comes down to the fact that an attacker needs to ”be” a user on the system, and then it is only the particular user which is compromised, ACRYLICS is claimed to be hard to attack from the outside.

It is of course very important for the security not to compromise with key derivation functions, random number generation, etc. We saw in Security Level 2, that the LUKS key Management [11] was constructed in a way, which reduced the possible key range significantly, and that would of course also weakness ACRYLICS if this is the case.

Security Level 3 marks the state of the current reference implementation. Security Level 4 includes the improvements for CAPM. As it is now claimed that ACRYLICS is hard to attack from the outside, we can turn our heads to the inside. The improvements in Section 5.6 concerns about malicious users in the system. As just discussed an attacker has to play the role as a user, and there ACRYLICS has to defend also againt malicious users. The challenges pointed out in Section 5.6 is of course a way to attack the system here in Security Level 3, but will improve ACRYLICS security to Security Level 4.

5.7.2 Security Level 4 - ACRYLICS with Improved CAPM

Security Level 4 is the best Security Level. In Security Level 3 ACRYLICS is claimed to be hard to attack from the outside and Security Level 4 aims to defend against insides attacks. As mentioned several times it is impossible to completely defend againt malicious users, as also said in Damgaard and Dupont [21]. However, some challenges with the CAPM scheme was improved in Section 5.6.

Security Level
Level 4: ACRYLICS With improved capability module
Level 3: ACRYLICS
Level 2: Full Disk Encryption
Level 1: No Disk Encryption

Figure 5.27: Security Level 4

Before the improvements, an malicious user, possibly an attacker would be able to access all shared files in the system. Even shared files not concerning the user. This is a security concern, and fortunately the shared secret improvement prevented this and a user can only access its own files and files shared with the user.

We have to note that a malicious user, possibly an attacker, can access files shared with the malicious user. As the files are shared with the user, the user will of course have access to the files. It is not possible to detect if a user is malicious or not, and if it was, we could as well just block the user completely from the system.

A classic example of a malicious user is when a employee is dismissed. Sometime this does not end well and the former employee wants to do harm. In such a case it is very important that the employees account is revoked completely. This is ensured with the seal improvement, which ensures that a revoked user can no longer decrypt shared data.

Of course the user account should be removed completely from the system, but it is of importance that a revocation to shared files can happen immediately and that it can be theoretical ensured.

It was also discussed in Section 5.6 that it might be possible secure even further for inside attack with the use of a secret KEY$_{CAPM}$ which could reside in a TPM or remote access control server. It would even be possible to also secure the system by using a secret stored key as the encrypting key normally derived form boot-password.

It is important to underline that the discussed challenges only is a problem if a user on the system is malicious, and the only thing other than the user’s own files it can access, is files explicit shared with the user. It would most likely also require physical access to the offline storage, as all programs also should be executed with the principle of least privileged.

VM FDE Bypass Attack vs. ACRYLICS

As the last thing we should discuss ACRYLICS in the worst case environment, recall the VM FDE bypass attack in Security Level 2, Subsection 4.4.2. We saw how multiple full disk encryption systems was completely bypassed.

ACRYLICS cannot either defend against this attack completely, but will not be completely bypassed right away. As guest host we have to assume that the host can read the value of our complete memory space at all time, hence the state of EM, hence the host does not need to know boot-password. But before the host can read any actual data on the disk a user needs to be logged in. When a user logs in the users encryption keys has to be stored somewhere in memory, and we need to assumed the the host also can read this. When a user is logged into ACRYLICS the hash table with capabilities is built. The host needs this table and the capabilities for each directory and file the host wants to access. We also have to assume the host can read this from memory.

It is now possible for the host system to access all the content that the user can access, it has the knowledge of EM's state, the user's encryption and the user's capability list.

But still is ACRYLICS not entirely broken, because the host cannot access any other data on the disk, so a user is first compromised when it is logged in and the user's encryption key is derived.

Over time we have to assume that any user has been active at some point, and in this case ACRYLICS is completely breached. But remember it is the case where the kernel memory can be observed constantly, and the host can extract any information it wants by inspection. Also two things should be noted. Firstly, no single super-user can access all data, hence every single user has to be compromised. Secondly, if the shared secret improvement was not made, then it would be possible for the host to read EM's state and use it to decrypt any shared file across the file system, but the improvement defend against this.

6 Implementation

In this section we will discuss some implementation specific details. In Section 6.1 the limitation of the implementation is discussed. In Section 6.2 the overall structs of the code is introduced, and how it is compiled. In Section 6.3 the specific and hardcoded choices are discussed. In Section 6.4 some of the important structs from the source code are introduced to give a better understating when examples of the running system are discussed in Chapter 7.

6.1 Limitations

The implementation of the complete ACRYLICS system is a huge task, which is why it has been necessary to make some limitations.

The development of ACRYLICS has been done on a Linux system, where a normal user program in Linux is used to simulate what ACRYLICS do. The execution environment is not finally designed and would require an entire capable operating system to run correctly. However simulation implementation shows how the boot process should work, how users are stored and the basic functionality of the system works. Since the prime focus in this thesis is to bind the security between operating system and file system this is the most important thing, and this is what the simulator program shows. In fact the simulator program can be seen as how ACRYLICS is mounted to a third party operating system, and still enforces the third party system to follow the specific design.

Only Symmetric Encryption

Since the implementation is completely free of library dependencies, an asymmetric key scheme has *not* been included. This means that the current implementation uses AES in all places where encryption is needed. Including where it should have been the asymmetric key pair that should have been used, recall the Hash-Key-Tree's Root node in Section 5.4. This limitation had to be accepted in order to reach an implementation state which shows the core values. The implementation is ready to store asymmetric keys in the correct places, and it is "only" an implementation of the asymmetric scheme itself that is missing.

Hash-Key-Tree's hashes are fixed

Hash values of sectors has been fixed, such that encrypted sectors have the hash value `0xcaca-caca-caca-caca-caca-caca-caca-caca` and plain text sectors have the hash values `0xaaaa-aaaa-aaaa-aaaa-aaaa-aaaa-aaaa-aaaa`. The primary reason is because of debugging the system. It is hard to debug a cryptographic system since all encrypted bytes just appear random. If just a simple mistake is made, a decryption can also result in something that looks like random data. To easier spot out the hash values, these were hardcoded. Code for constructing the real MAC (Message-Authentication-Code) is also missing, and the implementation is still in this state.

Key Derivation Function

In the current implementation the key derivation function is just a simple hash of the password, and must never be done like that in a production system. A real key derivation function is needed; a cryptographic secure hash function. However, it is “just” a matter of updating the logic in the key derivation function in the source code.

Free Sector Allocation

The ACRYLIC File System does not have implemented a proper free sector allocation scheme, and the current proof of concept allocator will only work for demonstrating purposes.

Hash-Key-Tree Leave Nodes Not Encrypted

The root of the Hash-Key-Tree is encrypted, although with the symmetric encryption as mentioned, but it supports the right boot process according to the ACRYLICS design. But leaf nodes are not encrypted as they should. Logical data sectors are encrypted with a key from the leaves, and the key is stored correctly, along with the hardcoded hash and the real Initialization Vector for the encryption. So the functionality works, but is not recursive to the root node. This means that the recursive encryption of the Hash-Key-Tree is not truly implemented, however, the leaf layer works as described in the design section and the root node is decrypted and checked on boot.

6.2 ACRYLICS Reference Implementation in Linux

The current implementation is running on Linux as a normal user program and is simulating ASCYLICS. A large normal file is functioning as the file system. When the program loads, it starts simulating ACRYLICS and is reading the file as the disk.

The source code is structured as follow

- **complete_test_system.c:** Contain user interaction and control the text user interface, hence is the User Module. This has code which is Linux specific. ACRYLICS integrated in an operating system should integrate with the User Interface from the operating system itself.
- **Acrylic_File_System:** Directory containing multiple source files which is the implementation of the entire ACRYLIC File System. Platform independent code, programmed in C with no dependencies.
- **Capability_Module:** Directory containing source code files for the Capability Module. Platform independent code, programmed in C with no dependencies.
- **Disk_Encryption_Module:** Directory containing source files for Encryption Module. Platform independent code, programmed in C with no dependencies.
- **Disk_Operations_Module:** Directory containing source code for disk operation. This is a platform dependent module and the code will be different depending on how a specific system or architecture writes to the disk. In Linux the code integrates with standard file I/O. In a barebone system, this module should contain code to talk with the physical disk.
- **Encryption_Interface** Directory containing the source code for an Encrypting Interface. As encryption functions can be used multiple places in an operating system, this is separated from the rest of the code and is made as an interface, so the interface both defines how to call cryptographic functions and contains the code itself.

  This means that the code for the AES encryption used in the system is implemented here. The current AES implementation used is a modified version of `https://github.com/kokke/tiny-AES-c`.This implementation is in the public domain and free to use. The modification made to the code has primarily been to dynamically allow different key-sizes, and the interface to calling the function has also been modified to contain information on key sizes.

The AES implementation is NOT time-attack-safe, which means that it should never be used in production systems as is.

The Encryption Interface only has one dependency, a platform dependency in term of the `encryption_interface_get_random_bytes`-function. Right now this is hardcoded to Linux rand() function, which is not cryptographic safe to use. The random function is seen as a platform dependency, since that platform, hence the Operating System has to offer a facility to get random bytes. This could for example be from the TPM (discussed in Section 4.1) or from the CPU itself. Intel for example has the RDRAND instruction. However, the choice of source to random bits must be done very carefully, since it must be really random and safe to use, to keep the system secure. There have been rumors that Intel's RDRAND contains possible backdoors [73].

- **Common and others:** In the common and in the source code root directory, there are some additional files containing commonly used code and some struct definitions which were necessary for simulating some kernel structures.

  The hash-table and the hash-functions in the common folder is code reused from the author's bachelor thesis [37].

In the code directory a bash file with name `build_for_linux.sh` is used to compile the system. The system should compile on most Linux systems and is compiled with the command

```
bash build_for_linux.sh
```

This will create the executable which is the complete simulation program. The file is called `acrylics`.

The program is executed with one parameter, which is the name of the file which simulates the disk.

```
./acrylics disk_file.img
```

If the file does not exist, it will be created and be formatted with the ACRYLIC File System.

As ACRYLICS is meant to be freestanding and integratable in a barebone operating system, it is designed to have no external dependencies. However, it uses some Linux specific calls as described. But the specific code has been properly separated such that it can be replaced and be target to compile against barebone platforms.

6.3 Implementation Specific Choices

AES is the only encryption scheme in the implementation and only the CBC mode of operation is used with a key size of 128 bits. So any encryption in the system use this. It is however also possible for a user to be non-encrypted.

The disk size is also fixed to contain 4096 physical blocks which are fixed at the size of 4096 bytes each. This gives a disk size of around 16MB. The logical sectors are fixed to a size of 8192 bytes each.

This will give a Hash-Key-Tree of height 1. Note that the Hash-Key-Tree of height 1 would support 4096 logical sectors (64 leaves with 64 entries), but with the fixed conditions there will only be a little less than 2048 logical sectors. The current implementation is not optimized and will build the complete leaf-layer, but only half of them will be used.

A system root user will always be hardcoded when starting a new disk. The username is `System_Root_User` and has the password `sysadmin`. The boot-password is also fixed to `sysadmin`.

6.4 Important Structures

In this section some of the most important structures from the source code are introduced. This will give a better understanding when examining the running system in Chapter 7 and give a good impression on how the theory is implemented in the code.

6.4.1 Encryption_Disk_Information

The `Encryption_Disk_Information` is the EM_h described Section 5.4. It has an encrypted and unencrypted part. So the main struct `Encryption_Disk_Information` contains the unencrypted part seen in Listing 6.2 and the encrypted part Listing 6.3.

```c
typedef struct Encryption_Disk_Information
{
    Encryption_Disk_Information_Unencrypted_Part unencrypted_part;
    Encryption_Disk_Information_Encrypted_Part encrypted_part;
} __attribute__((packed)) Encryption_Disk_Information;
```

Listing 6.1: Encryption_Disk_Information

Unencrypted Part

This is all the values needed for EM to know how to decrypt the second part and which location the system has on the disk. The structs size is 130 bytes, which we use to validate the functionality when examining the running system in Chapter 7.

```c
typedef struct Encryption_Disk_Information_Unencrypted_Part
{
     uint64_t  EM_version;
     uint8_t   em_header_iv[32];
     uint8_t   em_header_mac[16];
     uint64_t  LBA_em_this_info_end;
     uint64_t  LBA_first_logical_sector;
     uint64_t  LBA_last_logical_sector;
     uint64_t  last_logical_index;
     uint64_t  partition_LBA_start_sector;
     uint64_t  partition_LBA_last_sector;
     uint64_t  partition_pysical_sector_size;
     uint64_t  partition_logical_sector_size;
     uint16_t  symmetric_encryption_type;
     uint16_t  symmetric_encryption_algorithm;
     uint16_t  symmetric_encryption_mode_of_operation;
     uint32_t  symmetric_encryption_key_size;
} __attribute__((packed)) Encryption_Disk_Information_Unencrypted_Part;
```

Listing 6.2: Encryption_Disk_Information_Unencrypted_Part

Encrypted Part

The encrypted part holds information about EM. Does it have a cache system, does it have a journaling system etc. It also holds the important information of where is the start and end of the Hash-Key-Tree and more. The private and public key are also stored here, which is used to validate the root node of the Hash-Key-Tree. The hash for the root node is in the last part of the encrypted part with the encryption keys. As the encryption keys can vary in size, the `asymmetric_keys_private_public` is just a pointer to the end where multiple things happen. For instance the `Encryption_Module_Hash_Root_node_sector` -struct which holds the root node MAC. Struct is seen in Listing 6.4

```c
typedef struct Encryption_Disk_Information_Encrypted_Part
{
    uint64_t LBA_hash_tree_start;
    uint64_t LBA_hash_tree_end;
    uint64_t LBA_em_cache_start;
    uint64_t LBA_em_cache_end;
    uint64_t LBA_em_journaling_start;
    uint64_t LBA_em_journaling_end;
    uint64_t em_total_number_nodes;
    uint64_t em_tree_depth;
    uint64_t em_records_pr_sector;
    uint16_t asymmetric_encryption_type;
    uint16_t asymmetric_encryption_algorithm;
    int64_t  privat_key_size;
    int64_t  public_key_size;
    uint8_t  padding[19];
    // asymmetric_keys_private_public has to be at the end, since we allocate extra memory at the end for the keys.
    uint8_t  asymmetric_keys_private_public;
} __attribute__((packed)) Encryption_Disk_Information_Encrypted_Part;
```

Listing 6.3: Encryption_Disk_Information_Encrypted_Part

```c
typedef struct Encryption_Module_Hash_Root_node_sector
{
    uint8_t root_node_iv[32]; // TODO(Jørn) Should be removed when real asymmetric encryption is used.
    uint8_t root_node_MAC[64];

} __attribute__((packed)) Encryption_Module_Hash_Root_node_sector;
```

Listing 6.4: Encryption_Module_Hash_Root_node_sector

6.4.2 Hash-Key-Tree Node

As described in Section 5.4 a Hash-Key-Tree node contains 64 entries to childrens. An entry contains the random encryption key, MAC and IV for a child. In the code the struct is called `Disk_Encryption_Module_Expansion_Record` and is seen in Listing 6.5.

```c
typedef struct Disk_Encryption_Module_Expansion_Record
{
    uint8_t random_encryption_key[32]; // Max 256 bit key size.
    uint8_t record_mac[DISK_ENCRYPTION_MODULE_MAC_RECORD_SIZE];
    Encryption_Interface_AES_Initialization_Vector initial_vector;
} __attribute__((packed)) Disk_Encryption_Module_Expansion_Record;
```

Listing 6.5: Disk_Encryption_Module_Expansion_Record

6.4.3 User Master Table

To give an impression of the users, we will take a look at the User Master Table introduced in Section 5.3.

The table itself is hardcoded to the size of a logical sector. The essential is that it contains the `Master_User_Table_Entry` as also introduced in Section 5.3, and is seen in Listing 6.6.

```
typedef struct Master_User_Table
{
    Master_User_Table_Entry entry[HARDCODED_NUMBER_ENTRIES_MASTER_USER_TABLE];

} __attribute__((packed)) Master_User_Table;
```

Listing 6.6: Master_User_Table

The entries contains the username and a hash of the password, which are validated when the user attempts to log in to the system. The entry struct is shown in Listing 6.7, note which are the user's specific settings `Master_User_Table_Entry_Options`.

```
typedef struct Master_User_Table_Entry
{
  Master_User_Table_Entry_Options options;
  uint8_t  username[DEFAULT_MAX_USERNAME_LENGTH];
  uint8_t  password[128];
} __attribute__((packed)) Master_User_Table_Entry;
```

Listing 6.7: Master_User_Table_Entry

If the credentials are validated the user can have its own encryption settings as described in the design and the struct is shown in Listing 6.8. EM has to know the encryption settings, and they are also stored as a part of the entry.

```
typedef struct Master_User_Table_Entry_Options
{
    uint32_t  user_id;
    uint8_t  is_extended;
    uint32_t  password_size_bytes;
    Acrylic_Sector_Offset user_master_information_table;
    uint8_t   integrity_user_master_information_table[32];
    uint8_t  not_used[15];
} __attribute__((packed)) Master_User_Table_Entry_Options;
```

Listing 6.8: Master_User_Table_Entry_Options

6.4.4 Directory Table

To give an impression of the directory structures of the ACRYLIC File System, we will see the Directory Table (DT) in Listing 6.9 and the Directory Table Entries (DTE) in Listing 6.10. The structures functions as described in Section 5.3.

```
typedef struct Acrylic_Directory_Table
{
    Acrylic_Directory_Table_Entry entries[DEFAULT_TABLE_SIZE];
} __attribute__((packed)) Acrylic_Directory_Table;
```

Listing 6.9: Acrylic_Directory_Table

```c
typedef struct Acrylic_Directory_Table_Entry
{
    Acrylic_Object_ID     file_id;
    char                  object_name[256];
    uint16_t              entry_object_kind;
    uint8_t               unused_old_cap_block[42];
    uint8_t               integrigy_table_entry_object_index[32];
    Acrylic_Sector_Offset table_entry_object_index;
    uint8_t               unused[164];
} __attribute__((packed)) Acrylic_Directory_Table_Entry;
```

Listing 6.10: Acrylic_Directory_Table_Entry

6.4.5 Acrylics Object

The Acrylics Object (A-OBJ) was also introduced in Section 5.3. A-OBJ are the file meta data and hold a lot of information.

Interesting is the `Acrylic_Capability_Object_Block` which are the sensitive CAP-OBJ which has been described multiple times. If a file is shared, this is the exact struct that the users will point to. Each user will have their own CAP which are validated against the CAP-OBJ stores in `Acrylic_Capability_Object_Block`.

The implementation of the capability structures is very one-to-one with the introduced structures in Section 5.2.

```c
typedef struct Acrylic_Object
{
    Acrylic_Object_ID object_id;
    Acrylic_Sector_Offset table_backpointer;
    Acrylic_Sector_Offset table_entry_index;
    Acrylic_Sector_Offset this_secor_index;
    uint64_t owner_id;
    uint64_t size;
    Acrylic_Timestamp creation_date;
    Acrylic_Timestamp access_time;
    Acrylic_Timestamp modification_time;
    Acrylic_Capability_Object_Block capability_object_block;
    uint8_t unused[58];
    Acrylic_Data_Extent data_pointer[800];
} __attribute__((packed)) Acrylic_Object;
```

Listing 6.11: Acrylic_Object

7 Proof of Concept

In this section we will walk through some concrete examples of the running ACRYLICS reference implementation(**ARI**). The reference system was run as a program on Linux. Recall from Chapter 6 that the reference implementation is built in modules, the most of the modules is architecture independent, and can be compiled to any target. Besides the specific implemented UM for Linux, the only modules which are architecture specific are EM and DM. These have been specifically implemented to run on Linux, and it is therefore possible to run on Linux.

Running ACRYLICS In a Third Party Operating System

However it should be noted that ACRYLICS cannot guarantee security when running inside another system, recall the VM FED bypass attack discussion in Subsection 5.7.2. This also underlines the fact that ACRYLICS has to run baremetal in the operating system to ensure security, and the whole execution environment should also support and enforce the CAPM interface.

If programs were to be executed from ACRYLICS running in Linux, ACRYLICS has no way to tell Linux which capabilities a program has and Linux wouldn't know how to handle the capabilities and ensure the CAPM scheme is followed.

However, it is possible to test ACRYLICS as a mounted system, where the file system can be traversed with correct credentials, and can be modified if the schemes in ACRYLICS are followed strictly. This is actually a test case of the "offline storage"-security which was emphasized in the Security Level model. So this emphasizes that even if ACRYLICS is mounted to some third party operating system (in this case Linux), the third party operating system has to follow the schemes and the schemes cannot be circumvented, because of the encryption scheme.

Performance Test Considerations

Because of the implementation limitations described in Section 6.1, performance tests of the system will not be informative at this point. Capability-based systems have often been considered to have poor performance. However, modern hardware is extremely fast, and with better and better support for encryption in hardware, the encryption itself also becomes faster and faster. ACRYLICS is believed to not include significantly more encryption than other Full Disk Encryption solutions, and it is believed that it should be possible to achieve the same performance as existing solutions. As an example one could look at the performance comparison in CryFS [45].

In file systems free sector allocation is also a significant factor on performance as shortly discussed with ZFS in Subsection 4.3.2. Because ACRYLICS only have a proof-of-concept free sector allocation, this would also highly influence any performance measure and give inaccurate results.

7.1 Disk Formatting and System Setup

Before ACRYLICS can be used, it has to be initialized. ARI takes a file path as argument, where the file simulates the physical disk. If the file specified in the program arguments does not exist, ARI will create a file and initialize ACRYLICS in the file. If the file exists, ARI will boot with the information in the file.

In the initialization phase ACRYLICS needs a boot password. In ARI this has been hardcorded to ”sysadmin” which is also the password for the `System_Root`-user. When the boot password has been supplied the ACRYLICS system is set up. however, ACRYLICS require at least one user, such that it is possible to operate on the system and this user is hardcoded to be the `System_Root`-user with password ”sysadmin”. In a real scenario the operating system should prompt for this information. Some other information is hardcoded in ARI, like disk size, partition size and other details which are very important to ACRYLICS. Recall the EM_h described Section 5.4.

Remember that the first part of EM_h is unencrypted, since the data is needed to boot ACRYLICS and the second part is encrypted by the boot-password. Also recall that in ARI EM_h is the `Encryption_Disk_Information`-struct as described in Subsection 6.4.1.

EM’s Header Information

From the binary dump of the first part of the disk, it is possible to inspect and verify that the encrypted data starts at offset 0x00000082. The unencrypted part is 130 bytes as described in Subsection 6.4.1, and by inspection of the data in the unencrypted part, it can be verified that the system behave as expected. The second part is encrypted, and can only be verified if the boot-password is entered and ARI is inspected in a debugger.

```
00000000: e803 0000 0000 0000 05d8 36c5 b0e3 9199  ..........6.....
00000010: 4ff3 a74c 5fc0 c261 0000 0000 0000 0000  O..L_..a........
00000020: 0000 0000 0000 0000 caca caca caca caca  ................
00000030: caca caca caca caca 0100 0000 0000 0000  ................
00000040: 4400 0000 0000 0000 4400 0000 0000 0000  D.......D.......
00000050: de07 0000 0000 0000 0000 0000 0000 0000  ................
00000060: ff0f 0000 0000 0000 0010 0000 0000 0000  ................
00000070: 0020 0000 0000 0000 0200 0100 0200 8000  . ..............
00000080: 0000 b33a 28cd e830 0aef b2fa f0c7 18ba  ...:(..0........
00000090: a56a 5b37 3d63 ca8f 3264 c08d b099 1e2a  .j[7=c..2d.....*
000000a0: 64dc c504 4c62 93e5 0d97 98ca 2fe4 151b  d...Lb....../...
000000b0: 4002 d802 6be0 ed98 7588 b942 aec3 d88c  @...k...u..B....
000000c0: ff3f efc6 4dc2 e99b c27c 2690 eb01 517c  .?..M....|&...Q|
000000d0: 34bd 2834 fb6e 6e6b 7697 a592 3b82 839e  4.(4.nnkv...;...
000000e0: cab1 8708 88e5 e1c4 9dea 0922 7a33 e9c1  ..........."z3..
000000f0: 6d72 9f4e a652 0a69 488b 1b94 fce2 015d  mr.N.R.iH......]
00000100: 1fd3 f0b4 dac8 2fe5 4415 399d e80f 0d40  ....../.D.9....@
```

Hash-Key-Tree

In the setup process the Hash-Key-Tree is created and its size is calculated. With ARI’s hardcoded settings the tree should start at LBA 2. ARI’s settings are the same as illustrated in Figure 5.13, though only with 4096 physical sectors in total size, but the Hash-Key-Tree is structured the same way.

If we observe the partial binary dump, LBA 2 starting at offset 0x2000 to 0x2FFFF looks like random data and we assume that it is encrypted, like we expect the root node of the tree should be.

A couple of the implementation limitations can also be observed. The LBA 3 starting at 0x3000 is the first leafs of the Hash-Key-Tree, recall that the leafs in the implementation are not encrypted correctly. However, this makes it possible to see what is in the leaves. Remember the hash values for encrypted sectors is hardcoded with 0xca's, which we can verify in the binary dump. The other things in the record is the random encryption key and the IV used for the encryption for the correspondent sector. As also noted before, in the finished system, this will be encrypted such that it does not leaks any information.

```
00002fa0: 0f8a 1244 75e3 c56e b3d0 a46a 1c8f e5c3  ...Du..n...j....
00002fb0: 204d 5761 38e6 48eb 99bc 9dfc 03a3 61ba   MWa8.H.......a.
00002fc0: 231c 803b 8d06 c749 5e7f 6b9b b173 1561  #..;...I^.k..s.a
00002fd0: 92de 7916 1e44 8cfa 8d29 41e4 8d2b f8c5  ..y..D...)A..+..
00002fe0: ed62 1836 3f0e 8694 04b6 8678 4f27 8fa3  .b.6?......xO'..
00002ff0: 28b0 0767 9d76 900d e748 7f77 5840 ffc6  (..g.v...H.wX@..
00003000: 0000 0000 0000 0000 0300 0000 0000 0000  ................
00003010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003020: 1800 0000 0000 0030 001b c39a be70 2c67  .......0.....p,g
00003030: 0000 0000 0000 0000 ac53 c86d 3d7f 0000  .........S.m=...
00003040: caca caca caca caca caca caca caca caca  ................
00003050: 2d4b 2a54 32ef eb56 0f4a 973e 7d86 b6b1  -K*T2..V.J.>}...
00003060: caca caca caca caca 001b c39a be70 2c67  .............p,g
00003070: d05c f99d fc7f 0000 b7c0 ccb8 5b55 0000  .\..........[U..
00003080: caca caca caca caca caca caca caca caca  ................
00003090: e3f0 131e bb46 f30f 3a8d c6a0 8a13 6f61  .....F..:.....oa
```

ARI also writes to the last sector on the virtual disk, such that the file gets the right size. This forces Linux to allocate the whole size of the file. It also enables us to validate the last record used in the Hash-Key-Tree and we should be able to find the last record in the binary dump and validate. Since the simulated disk is only 4096 physical sectors, the last logical sector will be LSI 2014, remember that the size of the Hash-Key-Tree and EM's header data should be subtracted from the total disk size, before calculating the number of logical sectors. The last record should be at location offset 0x000227a0, which is validated in the binary dump shown in red.

```
00022770: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00022780: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00022790: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000227a0: 6b00 0000 7b6b df00 0f02 0000 0000 0000  k...{k..........
000227b0: 3020 0000 0000 0000 8000 0000 0000 0000  0 ..............
000227c0: caca caca caca caca caca caca caca caca  ................
000227d0: dba1 c677 60e5 050d a23e 357f 6951 9f6e  ...w`....>5.iQ.n
000227e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Start of Encrypted Data After Hash-Key-Tree

Note that the size of the Hash-Key-Tree is not optimized, hence it will always allocate all leaves for a layer, even though the last nodes will never be used. This also means that we should find the first LSI and the start of the ACRYLIC File System at LBA 68, which is confirmed here, marked in red, the first logical sector starts at offset 0x00044000.

```
00043fa0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00043fb0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00043fc0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00043fd0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00043fe0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00043ff0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00044000: e971 9a8f 6cd8 33d5 fd95 7020 116a 9f44  .q..l.3...p .j.D
```

```
00044010: a21e 48fc 3ae7 0f61 1b61 2962 6020 3876  ..H.:..a.a)b` 8v
00044020: bf54 2058 9267 44b5 a5d6 e9be 05a6 f5c2  .T X.gD.........
00044030: ab0a 90eb a69a 785d e026 1109 ee23 5557  ......x].&...#UW
00044040: e332 5084 32f7 6ab1 7cae 8045 693c 3994  .2P.2.j.|..Ei<9.
00044050: 3cda 316b e951 9408 7fef b766 0a06 8e8a  <.1k.Q.....f....
```

7.2 Boot Sequence and User Login

The boot-password and System_Root-user's credentials are loaded automatically and the system will boot as the System_Root-user. Recall the fixed password described in Section 6.3.

When the system is running, it is possible to change to another user with the change command in ARI. In the final system, the system should prompt for the boot-password and user login credentials. The system uses the boot-password to decrypt EM's encrypted header data and starts the encryption flow described in Section 5.5.

7.3 Object Creation and Sharing

We will now take a closer look at how ARI is working, and how capabilities are used. The examples in this section will use red to highlight important details and blue to highlight user input, which was entered into the system when the test was run.

User Creation

As the system only starts with the System_Root-user, the System_Root will start by creating a new user with user name worker and password 1234. This is done with the create-command and option 1. When the user has been successfully created the user list is printed, and we can see that there are two users in the system. Recall that the users are tightly bound to the ACRYLIC File System, and the new user information is stored at file system level and not just in a file in the operating system. The user information is encrypted with the settings of the encryption module.

Recall how the Master User Table is implemented described in Subsection 6.4.3. A free spot in MUT is found and the user credentials are stored. The selected encryption settings are also stored, such that the system can restore the encryption settings the next time the user logs in.

```
[SYSTEM] Print user list:
[0]: System_Root
[Acrylics](System_Root) : create
What do you want to create:
1: New User
2: New file
1
[SYSTEM] Create new user:
[SYSTEM] Enter user name: worker
[SYSTEM] Enter user password: 1234
[SYSTEM] Select personal encryption preferences:
1: AES-CBC-128
2: NO-Encryption
3: cancel
Enter: 2
[SUCCESS] User is created
[Acrylics](System_Root) : list
```

```
What do you want to list:
1: User list
2: Active users capabilities
3: Active users root directory
4: Malicious list directory
1
[SYSTEM] Print user list:
[0]: System_Root
[1]: worker
```

Note that the new user is created with no encryption. This will **not** affect the security of the stored user credentials nor the other users data, it will only affect the specific user's files. The rest of the section shows why.

Change Active User

First we need to change to the new user, this is possible with the `change` command. It will prompt for user credentials, and if the user credentials entered is found in Master User Table, the system will load the settings for that user, and this user will now be the active user.

```
[Acrylics](System_Root) : change
[SYSTEM] Enter username: worker
[SYSTEM] Enter user password: 1234
Password compare
00000000000070b17cde
00000000000070b17cde
[SUCCESS] Access granted
[Acrylics](worker) :
```

File Creation

Lets consider the `worker`-user is creating some files. This is done with the `create` command. `worker` selects option 2 to create a file and enters the name of the file. The system is now creating the file, generating the capability and gives back the owner capability, which is assigned to `worker`.

In this example `worker` is creating two files `/non-secret-file.txt` and `/another-file.txt`.

```
[Acrylics](worker) : create
What do you want to create:
1: New User
2: New file
2
[SYSTEM] Enter file name: /non-secret-file.txt
[FILE SYSTEM] Index found for file: 0
[SYSTEM] File: /non-secret-file.txt is created
[Acrylics](worker) : create
What do you want to create:
1: New User
2: New file
2
[SYSTEM] Enter file name: /another-file.txt
[FILE SYSTEM] Index found for file: 1
[SYSTEM] File: /another-file.txt is created
```

Recall that `worker` was created with non-encryption settings, this means that the user's files are not encrypted by default. An unencrypted user could be useful in a system which should obtain very high performance, and the information is not sensitive in any way. However, it should be noted that almost the tiniest information leakage can potentially give an attacker enough information to create some attack in the real world.

However, this non-encryption feature is also very good for debug purposes, because it gives us a unique insight into the system.

If we now examine `worker`'s root directory, we can see the two entries for the two files that `worker` created. Since the default settings for `worker` was non-encryption, the root directory is just stored in plain text. For a user with encryption enabled, we would of course only be able to see an encrypted sector. Note the gray color in the example which is an encrypted sector just before the plain text sector.

```
00053f80: eb9b e5ff ad10 b91e 3618 8370 39ab c904  ........6..p9...
00053f90: c9a0 1de2 bab4 2e66 6bd4 c625 b0a0 fe13  .......fk..%....
00053fa0: b7fb feb5 0ea5 bab1 23eb c4e5 3f95 c285  ........#...?...
00053fb0: 27ee 241d b7e1 363e 558f 48f5 ec4f ca27  '.$...6>U.H..O.'
00053fc0: 4ac7 c865 a998 347a 8d12 258a d36b ee20  J..e..4z..%..k.
00053fd0: a555 f867 1a38 727b c185 96a2 3a1f dd35  .U.g.8r{....:..5
00053fe0: 771e fc8b 6b31 4c12 c7ac 99a9 3677 31ff  w...k1L.....6w1.
00053ff0: 6059 d8e5 2b4c 043f e6f2 a708 0e41 34e9  `Y..+L.?.....A4.
00054000: 41fe baa3 80ff 4dbb 6e6f 6e2d 7365 6372  A.....M.non-secr
00054010: 6574 2d66 696c 652e 7478 7400 0000 0000  et-file.txt.....
00054020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00054030: 0000 0000 0000 0000 0000 0000 0000 0000  ................
.........  ....  ....  ....  ....  ....  ....  ....  ....  ................
000541e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000541f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00054200: 88ba 0be4 b049 fd87 616e 6f74 6865 722d  .....I..another-
00054210: 6669 6c65 2e74 7874 0000 0000 0000 0000  file.txt........
00054220: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Let's assume that `worker` is writing the string "This-is-non-secret-information" to `/non-secret-file.txt`.

```
[Acrylics](worker) : write
[SYSTEM] Enter line to write: This-is-non-secret-information
[SYSTEM] From file:
This-is-non-secret-information
```

If we search in the binary disk for the string "This-is-non-secret-information", the string can easily be found. Since the non-encrypted user is the owner, the data is not encrypted due to the settings

```
00057fa0: 2947 8a15 1e1b 9cad b943 2e64 1c32 210a  )G.......C.d.2!.
00057fb0: 0436 c2ce 9184 7863 34af e19f 3434 c856  .6....xc4...44.V
00057fc0: 53bf 7ad7 9bc3 10fd ca2f 2ab4 e18a 32ed  S.z....../*...2.
00057fd0: f84b 7371 452e 2ba3 1b9a b1a0 a885 e2e1  .KsqE.+.........
00057fe0: 25c7 f332 6b09 4dc7 8d22 6d27 77f9 25a0  %..2k.M.."m'w.%.
00057ff0: b9ea 980c ccc8 3aa7 87bd abed 106c 0f25  ......:......l.%
00058000: 88ba 0be4 b049 fd87 0000 0000 0000 0000  .....I..........
00058010: 0000 0000 0000 0000 0a00 0000 0000 0000  ................
00058020: 0000 0000 0000 0000 1f00 0000 0000 0000  ................
00058030: 4200 0000 0000 0000 0000 0000 f2a4 8f60  B..............`
00058040: 0000 0000 0000 0000 f2a4 8f60 0000 0000  ...........`....
00058050: 0000 0000 88ba 0be4 b049 fd87 0000 b927  .........I.....'
```

```
00058060: 12c4 c4f8 af1b 1bfa 2cc8 60ae b291 1000  ........,.`.....
00058070: 0100 1000 0100 ff00 0000 0000 0000 0000  ................
00058080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00058090: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000580a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000580b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000580c0: 401f 0000 0000 0000 0000 5468 6973 2d69  @.........This-i
000580d0: 732d 6e6f 6e2d 7365 6372 6574 2d69 6e66  s-non-secret-inf
000580e0: 6f72 6d61 7469 6f6e 0000 0000 0000 0000  ormation........
000580f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00058100: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Although `worker`'s data is not encrypted, the Hash-Key-Tree is still updated to preserve integrity over the whole disk. As mentioned earlier the hash values are hardcoded. With plain text data the hash value is 0xaa and 0xca for encrypted sectors.

If we look at the current state of the Hash-Key-Tree, we can see records for non-encrypted sectors interleaved with records for encrypted sectors and the Hash-Key-Tree doesn't really care if a sector is encrypted or not.

```
000031a0: 0102 0000 b000 0000 0000 0000 0000 0000  ................
000031b0: caca caca caca caca caca caca caca caca  ................
000031c0: caca caca caca caca caca caca caca caca  ................
000031d0: cf80 c57c d35d cae8 ab7f b95d b45f 6f97  ...|.].....]._o.
000031e0: 3020 0000 0000 0000 8000 0000 0000 0000  0 ..............
000031f0: 0102 0000 b000 0000 0000 0000 0000 0000  ................
00003200: caca caca caca caca caca caca caca caca  ................
00003210: e245 d23e 9523 22a0 80ea 5b25 af9d de1e  .E.>.#"...[%....
00003220: d023 96bf fd7f 0000 e845 b302 c855 0000  .#.......E...U..
00003230: 1024 96bf fd7f 0000 f06e 3e03 c855 0000  .$.......n>..U..
00003240: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
00003250: 004a 12d6 0100 0000 0a00 0000 0000 0000  .J..............
00003260: b7b1 b0e6 e27d 2ad0 7c3c f2c8 d4b0 7e50  .....}*.|<....~P
00003270: 2000 0000 0000 0000 0036 e46d 129e 6915   ........6.m..i.
00003280: caca caca caca caca caca caca caca caca  ................
00003290: 92e7 4879 59e1 009a dfba 3d60 ba8a 1b71  ..HyY.....=`...q
000032a0: 0000 0000 0000 0000 2055 12d6 e87f 0000  ........ U......
000032b0: 40b1 3e03 c855 0000 0063 12d6 e87f 0000  @.>..U...c......
000032c0: aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa  ................
000032d0: 2000 0000 0000 0000 2055 12d6 e87f 0000   ....... U......
000032e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000032f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00003300: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Remember that the Hash-Key-Tree should be encrypted in a final system, so this information would not be possible to see in the final system and would not leak any information whether a sector is encrypted or not, and in fact it does not either tell something about which encrypting setting that the given sector uses.

Share File With Reduced Capability

Let's see how capabilities are shared, in particular to share files, and how will it turn out if `System_Root` shares secret data with the non-encryption `worker`?

Let's assume that `System_Root` creates a secret file and shares it with `worker`. `System_Root` use `create` and selects the file name `/secret-file.txt`. When the file has been created `System_Root` uses the `write` and writes the string "secret-information" into the file.

```
[Acrylics](System_Root) : create
What do you want to create:
1: New User
2: New file
2
[SYSTEM] Enter file name: /secret-file.txt
[FILE SYSTEM] Index found for file: 1
[SYSTEM] File: /secret-file.txt is created
[Acrylics](System_Root) : write
[SYSTEM] Enter line to write: secret-information
[SYSTEM] From file:
secret-information
```

We saw with `worker`'s data, it was possible to find the data in plaintext in the binary disk, this is not the case now, since `system_root` is encrypting its files by default. If the data's sector location was tracked down, we would only see encrypted data.

Now `system_root` shares `READ/WRITE` capability to `worker`. This is done with the `share` command. `system_root` enters file name for the file to be shared, selects with option 3 that a `READ/WRITE` capability should be created. Since it is not possible to share a capability without the credentials for the user to be shared with, the credentials for `worker` has to be entered. It is a strong requirement since the user's capability table and root directory is encrypted with the key derived from the user's password. The system therefore needs the password to get the encryption key. When the system has the user's encryption key, the system can assign the newly created capability to the user. In this case to `worker`.

```
[Acrylics](System_Root) : share
[SYSTEM] Enter path on file to share: /secret-file.txt
[SYSTEM] Which permission would you like to share:
1: READ
2: WRITE
3: READ/WRITE
Enter number: 3
[SYSTEM] Share file: /secret-file.txt whit permission: 3
[SYSTEM] Enter user credentials to share with
[SYSTEM] Enter username: worker
[SYSTEM] Enter user password: 1234
Password compare
00000000000070b17cde
00000000000070b17cde
[SUCCESS] Access granted
Password compare
ae33efe3c0be562be3e6489fa8d2e01a
ae33efe3c0be562be3e6489fa8d2e01a
Access granted
[FILE SYSTEM] Index found for file: 2
```

Note the password comparison, which compares the Capability Object's password and the password from the user's capability, in this case if `System_Root` has access to the object. `System_Root` has access, and access is granted, and it is possible to share a reduced capability.

We change the active user to `worker` and let `worker` lists its root directory with the `list` command option 3.

```
Worker's root directory shows in plain text file name:
[Acrylics](worker) : list
What do you want to list:
1: User list
2: Active users capabilities
3: Active users root directory
4: Malicious list directory
3
[bb4dff80a3bafe41]: non-secret-file.txt
[87fd49b0e40bba88]: another-file.txt
[32c090b31fedf167]: secret-file.txt
```

We can now see the two files `worker` created earlier and the newly shared `secret-file.txt`. Note that `secret-file.txt` has Object ID `32c090b31fedf167` which corresponds to a capability with the same ID.

If `worker` uses the `list` command with option 2, `worker`'s capability list is printed. From that capability list we can see the shared capability for `secret-file.txt`.

```
[32c090b31fedf167]:
|-----------------|--------------------------------|
|E-Cap:           |                                |
|-----------------|--------------------------------|
|Capability id:   | 32c090b31fedf167               |
|-----------------|--------------------------------|
|Reduction field: |    ff    |    ff    |     3    |
|-----------------|--------------------------------|
|Password:        |7e89b83273c5152db56d0ece049b90c5|
|-----------------|--------------------------------|
|Class:           |0                               |
|-----------------|--------------------------------|
```

We should note several things here. Firstly we see that Object ID and Capability ID match. This is just a simple mechanism which makes efficient lookups possible. In the reduction field we see that the last (right most) sub-field's value is 3. This should be seen as the two least significant bits are ones. The least significant is the read bit and the next the write bit. The other sub-fields are flat as follows from the procedure of creating reduced capabilities.

Lastly note the capability's password, `7e89b83273c5152db56d0ece049b90c5`. The capability object's password is `0a3d3204a5583f8701b378cb8a32a320`. Now the capability password derivation function should reach the same password as in `worker`'s capability to grant access to read the file.

Read Data From File

By the `read` command, `worker` reads the `/secret-file.txt`-file, and as the correct password is derived access is granted and the string in the file has been decrypted and printed:

```
[Acrylics](worker) : read
[SYSTEM] Enter file to read: /secret-file.txt
0a3d3204a5583f8701b378cb8a32a320
d6f68b631ccc0ee7ae922e8d8801ebfa

```

```
Password compare
d6f68b631ccc0ee7ae922e8d8801ebfa
d6f68b631ccc0ee7ae922e8d8801ebfa
Access granted
[FILE DATA]: secret-information
```

The string "secret-information" is encrypted on the disk, and by the ACRYLICS design, this is still encrypted, even if the file is shared with a non-encrypted user. The encryption key is derived from the Hash-Key-Tree and EM's encryption key.

This does not only hold in this specific case, but holds for any case. This means that any user on the system theoretically can have their own encryption settings, but still be able to share files with each other and the owner has a guarantee that the file is encrypted safely with its encryption setting.

We should also look at an example of `worker` reduce its `READ/WRITE` capability, to another user. A new user is created called `worker2`. `worker` shares a reduced capability, like we saw before, but this time only with `READ` permission. If we change the active user to `worker2` and lists its capabilities, we will see the following capability.

```
[32c090b31fedf167]:
|-----------------|--------------------------------|
|E-Cap:           |                                |
|-----------------|--------------------------------|
|Capability id:   | 32c090b31fedf167               |
|-----------------|--------------------------------|
|Reduction field: |    ff    |     1    |     3    |
|-----------------|--------------------------------|
|Password:        |d6f68b631ccc0ee7ae922e8d8801ebfa|
|-----------------|--------------------------------|
|Class:           |0                               |
|-----------------|--------------------------------|
```

We should note that the Capability ID is the same, since it still corresponds to the same file with the same Object ID. This time we see that the middle sub-field has been reduced to 1 (only the least significant bit set). When CAPM logical ANDs the sub-fields together the result will be 1, hence `READ` only capability.

If we let `worker2` read the `/secret-file.txt`-file we will see that an extra step is taken in the derivation function, but still ends up with the correct password as in the capability, hence access granted.

```
[Acrylics](worker2) : read
[SYSTEM] Enter file to read: /double-share.txt
ae33efe3c0be562be3e6489fa8d2e01a
7e89b83273c5152db56d0ece049b90c5
d6f68b631ccc0ee7ae922e8d8801ebfa

Password compare
d6f68b631ccc0ee7ae922e8d8801ebfa
d6f68b631ccc0ee7ae922e8d8801ebfa
Access granted
[FILE DATA]: secret-information
```

Capability Validation Fails

The last example is a read request where the validation fails, hence access denied. If we assume `worker` tries the capability from before on another file, the derived password will not match the password from the capability and hence access will be denied.

`worker`'s Capability:

```
[32c090b31fedf167]:
|-----------------|--------------------------------|
|E-Cap:           |                                |
|-----------------|--------------------------------|
|Capability id:   | 32c090b31fedf167               |
|-----------------|--------------------------------|
|Reduction field: |    ff    |    ff    |     3    |
|-----------------|--------------------------------|
|Password:        |7e89b83273c5152db56d0ece049b90c5|
|-----------------|--------------------------------|
|Class:           |0                               |
|-----------------|--------------------------------|
```

Read attempt on another file:

```
[Acrylics](worker) : read
[SYSTEM] Enter file to read: /another-file.txt
570e234ca18f34346bbdbae448bfddd1
d6f68b631ccc0ee7ae922e8d8801ebfa

Password compare
d6f68b631ccc0ee7ae922e8d8801ebfa
7e89b83273c5152db56d0ece049b90c5
Access denied
[ABORT] Failed to read object
```

8 Conclusion and Future Work

8.1 Conclusion

The aim was to show if current operating systems core design makes them vulnerable, and if better IT-security could be obtained with a new core design which also can be proven secure.

To answer the questions relevant literature was introduced. Particularly Lopriore's [42] e-capabilities and Damgaard and Dupont's [21] Disk Encryption Scheme. The knowledge contributed to the analysis of the current systems, and the schemes were used in the core design of ACRYLICS.

In the analysis it was shown that some types of attacks are most likely impossible to defend against with the current core design given the use of ACL, super-user and security features which are not enforced. This answers the first question.

To quantify the analysis and later compare the analysed systems with ACRYLICS, security levels 1-4 were introduced. We saw that all the analysed systems without disk encryption are at Security Level 1 and with disk encryption they are at Security Level 2. ACRYLICS was proposed with a new core design with enforced capability-based access control and integrated disk encryption. ACRYLICS' security is implied by it following the proposed schemes which are proven secure. It was shown that ACRYLICS is at Security Level 3. ACRYLICS in Security Level 4 introduced stronger security for shared files with the shared secret improvement and defends against revoked users with the seal improvement.

ACRYLICS in the final Security Level 4 would prevent privilege escalation attacks since there is no real super-user which can access all files. This implies security but also privacy since no user can access another user's private files. Programs and services should only be granted the smallest set of capabilities, hence follow the principle of least privilege. So a malicious program or service should have even less power than the user executing it.

With the possible addition of a secure encryption key in CAPM, for example stored in the TPM or an external access control server, ACRYLICS might even prevent malicious users from forging more powerful CAPs or modify CAP-OBJs, but this can come with drawbacks and other security concerns.

A reference implementation of the core functionality in ACRYLICS was introduced. The reference implementation shows that it is possible to implement and run ACRYLICS in practice.

ACRYLICS presents a new core design for operating systems, it is built from schemes proven to be secure and in the Security Level model it is argued that better security is obtained than existing systems. ACRYLICS is the proposed solution to the second question.

8.2 Future Work

A lot of possibilities are seen in ACRYLICS and a lot of advanced features should be possible to implement given the design and attributes of the system. Firstly the execution environment should be designed. It should make use of the same CAPM scheme and continue to contribute to the enforced security between the operating system and file system, as has been obtained.

ACRYLICS is designed to be the core design in an Operating System, and with the final part of the execution environment, the system would be ready to test in a barebone operating system environment.

It should also be noted that a lot of existing capability based solutions make use of the ”pointer authentication mechanism”, which Apple also uses. This approach should be explored, and see if it could be combined with the already existing CAPM scheme in ACRYLICS.

ACRYLICS aims for high security and protection of users, and the fact that users in between are safe for one another, including from the system administrator, it would be a nice feature to integrate ”deniable data”. File systems with this attribute are known as ”deniable file systems”, and makes it possible to hide data which can not be proven to exist. Given the design of ACRYLIC there should be a large potential to integrate this, and ensure the deniability. Since a user is not capable of decrypting the entire disk anyway in the current design, it is possible to hide data and claim it is an unreachable part of the disk for the given user.

As we have seen, root-of-trust is a major concern in current systems, and the proposal with a secure CAPM encryption key stored in the TPM or on an access control server should be investigated. Furthermore it could be investigated if ACRYLICS could function as a distributed file system, as the permissions should be safe cross systems. In this way a user could log into a workstation and get its credentials validated and gets his capability table, maybe directly from some access control server. With the capabilities it would be possible to ask the remote file server for files, and only with the correct capabilities the files can be retrieved.

Bibliography

[1] *About encrypted storage on your new Mac.* https://support.apple.com/en-us/HT208344. Accessed: 2021-05-11.

[2] Mike Accetta et al. "Mach: A new kernel foundation for UNIX development". In: *Proceedings of Summer Usenix.* July 1986.

[3] Mamdouh Alenezi and Mohammad Zarour. "On the Relationship between Software Complexity and Security". In: *CoRR* abs/2002.07135 (2020). arXiv: 2002.07135.

[4] *Apple iMac Pro and Secure Storage.* https://duo.com/blog/apple-imac-pro-and-secure-storage. Accessed: 2021-05-11.

[5] *Apple Platform Security - Hardware security overview.* https://support.apple.com/en-gb/guide/security/secf020d1074/1/web/1. Accessed: 2021-05-04.

[6] *Apple Platform Security - Secure Enclave overview.* https://support.apple.com/en-gb/guide/security/sec59b0b31ff/web. Accessed: 2021-05-04.

[7] *Apple Platform Security - System security overview.* https://support.apple.com/en-gb/guide/security/sec114e4db04/1/web/1. Accessed: 2021-05-04.

[8] *Application Sandbox.* https://source.android.com/security/app-sandbox. Accessed: 2021-05-19.

[9] Leonardo Bertolin Furstenau et al. "20 Years of Scientific Evolution of Cyber Security: a Science Mapping". In: *Proceedings of the International Conference onIndustrial Engineering and Operations Management.* Apr. 2020.

[10] Saikat Biswas et al. "A study on remote code execution vulnerability in web applications". In: *International Conference on Cyber Security and Computer Science (ICONCS 2018).* 2018.

[11] Simone Bossi and Andrea Visconti. "What Users Should Know About Full Disk Encryption Based on LUKS". In: *Cryptology and Network Security - 14th International Conference, CANS 2015, Marrakesh, Morocco, December 10-12, 2015, Proceedings.* Vol. 9476. Lecture Notes in Computer Science. Springer, 2015, pp. 225–237.

[12] I. T. Bowman, R. C. Holt, and N. V. Brewster. "Linux as a case study: its extracted software architecture". In: *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002).* 1999, pp. 555–563.

[13] R. Canetti. "Universally composable security: a new paradigm for cryptographic protocols". In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science.* 2001, pp. 136–145.

[14] R. Capizzi et al. "Preventing Information Leaks through Shadow Executions". In: *2008 Annual Computer Security Applications Conference (ACSAC).* 2008, pp. 322–331.

[15] *Capsicum-Linux commit historic.* https://github.com/google/capsicum-linux/commits. Accessed: 2021-05-19.

[16] Eoghan Casey and Gerasimos J. Stellatos. "The Impact of Full Disk Encryption on Digital Forensics". In: *SIGOPS Oper. Syst. Rev.* 42.3 (Apr. 2008), pp. 93–98.

[17] M. D. Castro, R. Pose, and C. Kopp. "Password-Capabilities and the Walnut Kernel". In: *Comput. J.* 51 (2008), pp. 595–607.

[18] E. S. Chang et al. "Managing cyber security vulnerabilities in large networks". In: *Bell Labs Technical Journal* 4.4 (1999), pp. 252–272.

[19] Omar Choudary, Felix Gröbert, and Joachim Metz. "Infiltrate the Vault: Security Analysis and Decryption of Lion Full Disk Encryption." In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 374.

[20] *Crouching T2, Hidden Danger.* https://ironpeak.be/blog/crouching-t2-hidden-danger/. Accessed: 2021-05-11.

[21] Ivan Damgård and Kasper Dupont. "Universally Composable Disk Encryption Schemes." In: *IACR Cryptology ePrint Archive* 2005 (Jan. 2005), p. 333.

[22] Luke Deshotels et al. "IOracle: Automated Evaluation of Access Control Policies in IOS". In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security.* ASIACCS '18. Incheon, Republic of Korea: Association for Computing Machinery, 2018, pp. 117–131.

[23] Morozov Dmitry and Ponomareva Elena. "Linux Privilege Increase Threat Analysis". In: *2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT).* 2020, pp. 0579–0581.

[24] Kamy Farahbod, Conrad Shayo, and Jay Varzandeh. "Cybersecurity indices and cybercrime annual loss and economic impacts". In: *Journal of Business and Behavioral Sciences* 63 (2020).

[25] Kristian Gjøsteen. "Security notions for disk encryption". In: *European Symposium on Research in Computer Security.* Springer. 2005, pp. 455–474.

[26] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. "A capability-based security approach to manage access control in the Internet of Things". In: *Mathematical and Computer Modelling* 58.5 (2013). The Measurement of Undesirable Outputs: Models Development and Empirical Analyses and Advances in mobile, ubiquitous and cognitive computing, pp. 1189–1205.

[27] Hassan El-Hadary and Sherif El-Kassas. "Capturing security requirements for software systems". In: *Journal of Advanced Research* 5.4 (2014), pp. 463–472.

[28] *HAFNIUM targeting Exchange Servers with 0-day exploits.* https://www.microsoft.com/security/blog/2021/03/02/hafnium-targeting-exchange-servers/. Accessed: 2021-05-20.

[29] Ian Haken. *Bypassing Local Windows Authentication to Defeat Full Disk Encryption.* https://www.blackhat.com/docs/eu-15/materials/eu-15-Haken-Bypassing-Local-Windows-Authentication-To-Defeat-Full-Disk-Encryption-wp.pdf. Accessed: 2021-05-31. 2015.

[30] J. Alex Halderman et al. "Lest We Remember: Cold-Boot Attacks on Encryption Keys". In: *Commun. ACM* 52.5 (May 2009), pp. 91–98.

[31] Norm Hardy. "The Confused Deputy: (or why capabilities might have been invented)". In: *ACM SIGOPS Operating Systems Review* 22.4 (1988), pp. 36–38.

[32] Y. Hebbal. "Bypassing Full Disk Encryption with Virtual Machine Introspection". In: *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA).* 2019, pp. 1–8.

[33] Gernot Heiser. *The seL4.* https://sel4.systems/About/seL4-whitepaper.pdf. Accessed: 2021-05-31. 2020.

[34] Gernot Heiser and Kevin Elphinstone. "L4 Microkernels: The Lessons from 20 Years of Research and Deployment". In: *ACM Trans. Comput. Syst.* 34.1 (Apr. 2016).

[35] G. Hernandez et al. "BigMAC: Fine-Grained Policy Analysis of Android Firmware". In: *USENIX Security Symposium.* 2020.

[36] Ayelet Israeli and Dror G. Feitelson. "The Linux kernel as a case study in software evolution". In: *Journal of Systems and Software* 83.3 (2010), pp. 485–501.

[37] Patrick Jakobsen, Jørn Guldberg, and Jakob Kjær-Kammersgaard. *Bachelor Project in Compiler Construction: Kitty.* University of Southern Denmark. BA project. May 2019.

[38] Poul-Henning Kamp. "GBDE-GEOM based disk encryption". In: *BSDCon 2003.* Jan. 2003.

[39] Gerwin Klein et al. “SeL4: Formal Verification of an OS Kernel”. In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles.* SOSP ’09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220.

[40] Jesse D. Kornblum. “Implementing BitLocker Drive Encryption for forensic analysis”. In: *Digital Investigation* 5.3 (2009), pp. 75–84.

[41] R. Levin et al. “Policy/Mechanism Separation in Hydra”. In: *Proceedings of the Fifth ACM Symposium on Operating Systems Principles.* SOSP ’75. Austin, Texas, USA: Association for Computing Machinery, 1975, pp. 132–140.

[42] Lanfranco Lopriore. “Access right management by extended password capabilities”. In: *International Journal of Information Security* 17 (Oct. 2018).

[43] Junaid M. et al. “Security Flaws of Operating System Against Live Device Attacks: A case study on live Linux distribution device”. In: *2019 Sixth International Conference on Software Defined Systems (SDS).* June 2019, pp. 154–159.

[44] Derek Manky. “Cybercrime as a service: a very modern business”. In: *Computer Fraud & Security* 2013.6 (2013), pp. 9–13.

[45] Sebastian Messmer. *CryFS: Design and Implementation of a Provably Secure Encrypted Cloud Filesystem.* https://www.cryfs.org/cryfs_mathesis.pdf. Accessed: 2021-05-31. 2015.

[46] Sebastian Messmer et al. “A Novel Cryptographic Framework for Cloud File Systems and CryFS, a Provably-Secure Construction”. In: *Data and Applications Security and Privacy XXXI.* Ed. by Giovanni Livraga and Sencun Zhu. Cham: Springer International Publishing, 2017, pp. 409–429.

[47] *Microsoft - Compare Windows 10 editions.* https://www.microsoft.com/en-us/WindowsForBusiness/Compare. Accessed: 2021-05-11.

[48] *Microsoft - File System Functionality Comparison.* https://docs.microsoft.com/en-us/windows/win32/fileio/filesystem-functionality-comparison. Accessed: 2021-05-20.

[49] *Microsoft - Secure boot.* https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot. Accessed: 2021-05-10.

[50] *Microsoft - Virtual Secure Mode.* https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/vsm. Accessed: 2021-05-11.

[51] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. *Capability myths demolished.* Tech. rep. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003.

[52] Dan Mossop and Ronald Pose. “Information Leakage and Capability Forgery in a Capability-Based Operating System Kernel”. In: *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops.* Ed. by Robert Meersman, Zahir Tari, and Pilar Herrero. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 517–526.

[53] R. M. Needham and R. D.H. Walker. “The Cambridge CAP Computer and Its Protection System”. In: *SIGOPS Oper. Syst. Rev.* 11.5 (Nov. 1977), pp. 1–10.

[54] *New McAfee Report Estimates Global Cybercrime Losses to Exceed $1 Trillion.* https://www.mcafee.com/enterprise/en-us/about/newsroom/press-releases/press-release.html?news_id=6859bd8c-9304-4147-bdab-32b35457e629. Accessed: 2021-05-27.

[55] Andy Ozment and Stuart E Schechter. “Milk or wine: does software security improve with age?” In: *USENIX Security Symposium.* Vol. 6. 2006.

[56] *POSIX file capabilities, the dark side.* https://blog.sevagas.com/POSIX-file-capabilities-the-dark-side. Accessed: 2021-05-07.

[57] John S. Quarterman, Abraham Silberschatz, and James L. Peterson. “4.2BSD and 4.3BSD as Examples of the UNIX System”. In: *ACM Comput. Surv.* 17.4 (Dec. 1985), pp. 379–418.

[58] *ROCA: Infineon TPM and Secure Element RSA Vulnerability Guidance.* https://www.ncsc.gov.uk/guidance/roca-infineon-tpm-and-secure-element-rsa-vulnerability-guidance. Accessed: 2021-05-07.

[59] Richard Russon and Yuval Fledel. *NTFS documentation.* https://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf. Accessed: 2021-05-31. 2004.

[60] J. H. Saltzer. "Protection and the control of information sharing in multics". In: *Commun. ACM* 17 (1974), pp. 388–402.

[61] Bruce Schneier. *Software complexity and security.* https://www.schneier.com/essays/archives/1999/11/a_plea_for_simplicit.html. Accessed: 2021-05-31. 2000.

[62] *Secure Boot in the Era of the T2.* https://duo.com/labs/research/secure-boot-in-the-era-of-the-t2. Accessed: 2021-05-11.

[63] *Security certifications for the Secure Enclave Processor.* https://support.apple.com/en-gb/guide/sccc/sccca7433eb89/web. Accessed: 2021-05-04.

[64] Jianxiong Shao et al. "Formal Analysis of Enhanced Authorization in the TPM 2.0". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.* ASIA CCS '15. Singapore, Republic of Singapore: Association for Computing Machinery, 2015, pp. 273–284.

[65] *Shut the HAL Up.* https://android-developers.googleblog.com/2017/07/shut-hal-up.html. Accessed: 2021-05-17.

[66] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10th Edition.* Wiley, 2018.

[67] Amit Singh. *Mac OS X Internals: A Systems Approach (paperback).* Addison-Wesley Professional, 2006.

[68] *Sniff, there leaks my BitLocker key.* https://labs.f-secure.com/blog/sniff-there-leaks-my-bitlocker-key/. Accessed: 2021-05-10.

[69] D. A. Solomon. "The Windows NT kernel architecture". In: *Computer* 31.10 (1998), pp. 40–47.

[70] Jeff Vander Stoep and Sami Tolvanen. *Linux Security Summit Aug 2018 - Android Kernel Security.* https://events19.linuxfoundation.org/wp-content/uploads/2017/11/LSS2018.pdf. Accessed: 2021-05-19.

[71] Andrew Tanenbaum. "Lessons learned from 30 years of MINIX". In: *Communications of the ACM* 59 (Feb. 2016), pp. 70–78.

[72] Ken Thompson and Dennis M Ritchie. "The UNIX time-sharing system". In: *Communications of the ACM* 17.7 (1974), pp. 365–375.

[73] *Torvalds shoots down call to yank 'backdoored' Intel RdRand in Linux crypto.* https://www.theregister.com/2013/09/10/torvalds_on_rrrand_nsa_gchq. Accessed: 2021-05-29.

[74] Wiem Tounsi and Helmi Rais. "A survey on technical threat intelligence in the age of sophisticated cyber attacks". In: *Computers & Security* 72 (2018), pp. 212–233.

[75] *TPM 2.0 Library.* https://trustedcomputinggroup.org/resource/tpm-library-specification/. Accessed: 2021-05-07.

[76] Sven Türpe et al. "Attacking the BitLocker Boot Process". In: *Trusted Computing.* Ed. by Liqun Chen, Chris J. Mitchell, and Andrew Martin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 183–196.

[77] Sven Vermeulen. *SELinux and extended permissions.* https://blog.siphos.be/2017/11/selinux-and-extended-permissions/. Accessed: 2021-05-19.

[78] Rossouw von Solms and Johan van Niekerk. "From information security to cyber security". In: *Computers & Security* 38 (2013). Cybercrime in the Digital Economy, pp. 97–102.

[79] Robert NM Watson et al. “Capsicum: Practical Capabilities for UNIX.” In: *USENIX Security Symposium*. Vol. 46. 2010, p. 2.

[80] Z. Weinberg et al. “I Still Know What You Visited Last Summer: Leaking Browsing History via User Interaction and Side Channel Attacks”. In: *2011 IEEE Symposium on Security and Privacy*. 2011, pp. 147–161.

[81] Richard Wilkins and Brian Richardson. “UEFI secure boot in modern computer security solutions”. In: *UEFI Forum*. 2013.

[82] *Windows 10 Still Won’t Let You Use These File Names Reserved in 1974*. https://www.howtogeek.com/fyi/windows-10-still-wont-let-you-use-these-file-names-reserved-in-1974/. Accessed: 2021-05-27.

[83] Bob Zeidman. “A Code Correlation Comparison of the DOS and CP/M Operating Systems”. In: *Journal of Software Engineering and Applications* 7 (May 2014).

[84] Bob Zeidman. “Source Code Comparison of DOS and CP/M”. In: *Journal of Computer and Communications* 04 (Jan. 2016), pp. 1–38.