CHALMERS

Digital Right Management on embedded systems

Protecting video content in players

Master of Science Thesis in Secure and Dependable Computer Systems

ALBAN DIQUET

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, November 2009

Digital Right Management on embedded systems
Protecting video content in players

ALBAN DIQUET

Examiner: BJÖRN VON SYDOW

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden November 2009

I. INTRODUCTION 6

I.1. About Digital Right Management 6
I.2. Sigma Designs 7
I.3. Purpose of the Thesis 7
I.4. The Secure Media Processor Architecture 8
I.5. Content of the report 9

II. PROJECT 1 INVESTIGATION OF THE DIFFERENCES BETWEEN DIFFERENT VERSIONS OF SIGMA BOARDS 11

II.1. Introduction 11
II.2. Project goals 11
II.3. Analysis 12
II.4. Work accomplished 13
II.5. Results 18

III. PROJECT 2 IMPLEMENTATION OF A NEW DRM SPECIFICATION 19

III.1. Background 19
III.2. Project goals 19
III.3. Work accomplished 23
III.4. Results 26

IV. PROJECT 3 CPU-XPU COMMUNICATION PERFORMANCE MEASURE 27

IV.1. Background 27
IV.2. Work accomplished 28
IV.3. Results on tango2 30
IV.4. Results on tango3 31

V. PROJECT 4 SERIAL FLASH ACCESS LIBRARY 33

V.1. Background 33
V.2. Work Accomplished 34
V.3. Problems Encountered 38

VI. PROJECT 5 KEY-LOADING TOOLS 39

VI.1. Background 39
VI.2. Work accomplished 41
VI.3. Results 48
VI.4. Problems encountered 49

VII. CONCLUSION 51

VIII. APPENDIX 52

VIII.1. PROJECT 1 52
VIII.2. PROJECT 3 54

Acknowledgements

I am grateful to Sigma Designs Inc for giving me the opportunity to do this master thesis in Milpitas, California. I would like to thank Fabrice Gautier, my supervisor at Sigma Designs for making me very welcome in the team, and for the help he gave me during this thesis.

I would like also to thank the University of Chalmers for the “Secure and Dependable Computer System” master program, my examiner Prof. Björn Von Sydow, and the director of the master program Dr Roger Johansson.

I. Introduction

I.1. About Digital Right Management

Digital rights management (DRM) is a generic term for access control technologies that can be used by copyright holders in order to impose limitations on the usage of digital content and devices. DRM systems are used by most of the entertainment industry: music, films, video games and any other media that can be digitized and passed around.

Digital Right Management has been developed in response to the rapid increase in piracy of commercially marketed material, especially through the widespread use of the Internet and peer to peer networks. The industry wants to prevent people from being able to copy, and freely share digital content without paying rights.

For example, DRM systems used on audio CDs prevent users from being able to copy protected CDs. It's only possible to listen to such CDs, because there is specific code embedded on them that prevents copying.

In the film industry, DRM systems are widely used too.
Content Scrambling System (CSS) is an early example. It is the content protection employed on film DVDs since 1996. CSS relies on an encryption algorithm, so that the film is encrypted on the DVD, and a DVD player has to be able to decrypt in order to play it on a TV, for example.

A more recent example is the Advanced Access Content System (AACS), used to protect HD DVD and Blu-Ray Discs. It has been developed by a consortium that includes Disney, Intel, Microsoft, Warner Brothers, IBM, Toshiba and Sony, and like CSS, relies on an encryption of the film.

On a side note, both CSS and AACS protections have been broken because hackers were able to find secret keys in players, and use them to decrypt and remove the protection of any film, making it possible to copy the film. Those keys also have been then released on the Internet.

I.2. Sigma Designs

Sigma Design Inc is a company based in Milpitas, California.
It develops System-On-a-Chip semiconductors called Secure Media Processors, for consumer products such as DVD players, IPTV set-top boxes, Blu-ray players/recorders, HDTVs, Portable Media Players...

Those Secure Media Processors provide solutions for products requiring audio/video processing (like MPEG or DVD decoding). They also offer advanced content protection, and support a variety of Digital Rights Management (DRM) and Conditional Access (CA) solutions.

I joined Sigma Design as a member of the software development team, specialized in the security aspect, which is content protection and DRM systems.

I.3. Purpose of the Thesis

Today most of the video content someone can buy (dvd, blue-ray discs, etc…), is protected with a DRM system to prevent copy. However, video players have to be able to play protected films.

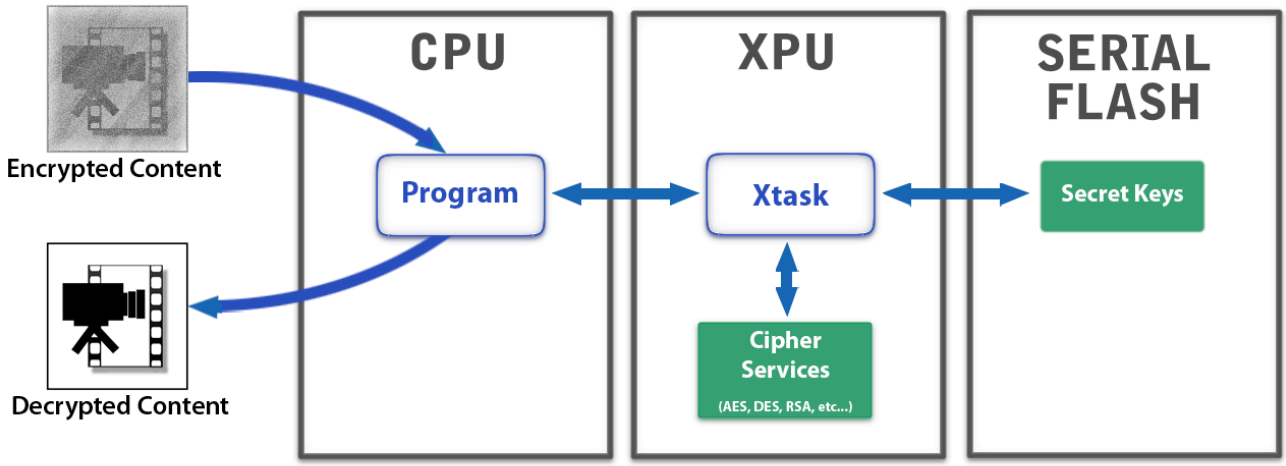
For this reason, the boards Sigma sells have to provide special security features so that:
They can comply with DRM specifications in order to be able to decrypt protected content (films).
They are able to protect the secrets stored on the board (like secret keys).

This report is about how the video players are designed, and programmed, to provide such features.

I.4. The Secure Media Processor Architecture

Sigma boards are designed around an architecture called the Secure Media Processor architecture.

Figure 1: The Secure Media Processor Architecture



Sigma has two families of Secure Media Processors, codenamed Tango2 and Tango3. They include a Secure Processor called XPU. The XPU runs a small Operating System called xos. DRM implementations are written as programs running in the XPU. They are called Xtasks.

Because most DRM systems for video content rely on encryption algorithms, a number of cryptographic schemes (like AES, RSA, etc..) are present in the chip to protect the contents (like audio, movies etc...) from unauthorized copying. In addition to providing a secure boot process, the XPU supports the secure execution of cryptographic schemes like Digital Rights Management (DRM) and Conditional Access (CA) software.

A determined hacker using specialized equipment might be able to hack into the CPU, which runs the main applications. Therefore, all DRM cryptographic services are run in the XPU. The XPU resides in a privileged hardware block of the chip, and runs these security-related softwares.
Highly confidential DRM information (certificates, private keys, etc...) are stored on the internal 64KB serial flash accessible only from the XPU.

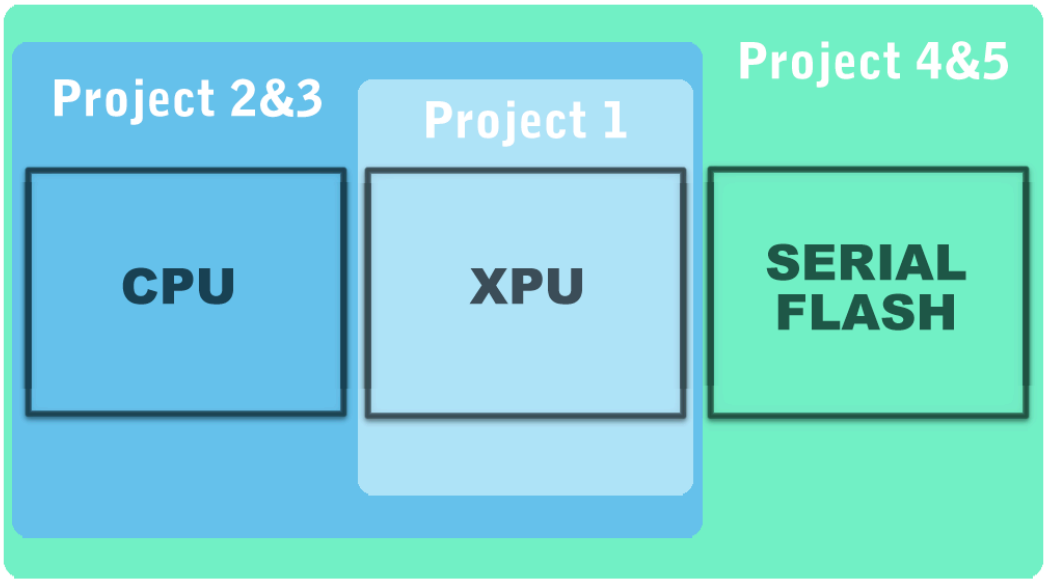
A xtask is a program that runs on the XPU and provides DRM cryptographic services to the main application running on the CPU. The cryptographic services may include authentication and steady-state decryption phases, as dictated by a given DRM. The xtasks do not give the DRM secrets to the main application. Rather, they use the DRM secrets on behalf of the main application.

To avoid unauthorized third parties from writing their own xtasks or altering existing xtasks, every xtask is signed by Sigma and optionally encrypted.

I.5. Content of the report

This reports present five different projects, all related to the Secure Media Processor architecture, and Sigma boards. It is the work I did during the six months of the thesis, and is about adding or improving features to the Sigma boards, by programming the architecture.

Figure 1: Projects presented in this report, and their relation with the Secure Media Processor architecture



As shown in the figure, I first worked on the XPU only, then on the XPU–CPU pair, and finally on the whole architecture, including the XPU, the CPU, and the serial flash.

Here is a small description of those five projects:

Project 1: Investigating Sigma boards and finding the differences between tango2 and tango3 boards.

Project 2: Writing support for a new DRM specification.

Project 3: Measuring the time it takes for the CPU and the XPU to communicate with each other.

Project 4 & 5: Creating tools to write and read data (keys) to the serial flash.

II. Project 1

Investigation of the differences between different versions of Sigma boards

II.1. Introduction

As stated before, Sigma currently has two families of Secure Media Processors, codenamed Tango2 and Tango3. They include a Secure Processor called XPU that runs a small Operating System called xos.

Two versions of the Secure OS exist, xos which is used on Tango2 boards, and xos2 which is used on Tango3 boards. They are different enough so that a certain amount of work is required when porting a xtask from xos to xos2.

II.2. Project goals

The main goal of the project is to provide a library that will ease the porting effort between the different chips and xos versions (xos on Tango2 / xos2 on Tango3).

In order to do this, two steps were defined:

Analyze and define the abstraction layer

Looking at differences and similarities between xos and xos2.
Classifying the differences into categories... (cipher services, memory, debug...)
Looking for places where the code differs between the xos and xos2 ports of existing xtasks.
Defining the function provided by abstraction layer.

Implementation

Implementing the library for tango2.
Implementing the library for tango3.
Writing test xtask code using this abstraction layer.

II.3. Analysis

The first and main part of this project was to analyze existing xtasks and to understand how they worked. Xtasks are basically a C program with specific types, functions, and libraries, related to the operating system running on the board, xos.
So, I could easily understand the syntax because it's C language, but had to decipher the code itself in order to understand what the xtask was doing and how it was done.

While doing this, I also started to look at the differences between tango2 and tango3 xtasks. Three things helped me to find and understand the differences: the source codes of existing xtasks, the twiki website, and a meeting with other members of the software development team.

II.3.1. Xtask source codes

In general, every xtask is initially written to work on tango2, and is later ported on tango3. So, both versions of the xtask have the same function, which make it easier to spot the differences in the code.
I had to look at the code of existing xtasks. First the "hello-world" xtask, and then more complicated ones. I wrote down all the differences I could see between the tango2 and tango3 implementations of the same xtask, and separated those differences into categories.

II.3.2. Wiki articles

The software department uses a "wiki" website that contains many articles written by members of the software team, about various topics related to Sigma boards. I could find articles concerning the xos and xos2 kernels, and also an article giving insight on how to port a xtask from tango2 to tango3.

II.3.3. Conclusion of the analysis

From this analysis I drew a list of all the differences I could find in the code: functions with a new name, new modes of operation, different functionalities.
Finally, a meeting with other members of the software team helped me to find out what were actually the trickiest parts when porting a xtask, as some differences are just trivial to handle.

The main problems when porting from tango2 to tango3 are:

- the memory configuration: size, partition, attribution, and mapping.
- the cipher services: mode of operation, functionalities.

The memory is the most complex problem when porting, as the addresses manipulated in the xtask are not the same between tango2 and tango3. If the xtask being ported manipulates addresses, the software developer has to rethink it entirely.
However, a library cannot take care of memory addressing because it's just too complicated to be handled automatically.

Hence, it was decided that my goal for this project would be to write a library that would ease the porting of xtask code related to cipher operations, which is second main problem when writing a port. Then, I would have to write the actual library.

II.4. Work accomplished

After defining the concrete goals of the project in the analysis part, I then had to start working on it: a library that would ease the porting effort of cipher operations between tango2 and tango3. I had to understand of the cipher operations work and what are the exact differences between tango2 and tango3.

II.4.1. Mode of operation of the cipher services

The general way of using cipher services on both tango2 and tango3 boards, is to first fill a configuration structure containing:

an ID representing the cipher operation to do (AES encryption/decryption, DES encryption/decryption, SHA1, etc...)

a source address (what to encrypt or decrypt)

a destination address (where to put the result), that can be the same as a the source address

a size in bytes (how many bytes should be processed)

256 bits used to store a key

256 bits used to store an initialization vector

This structure is then given as an argument to function that triggers the right cipher operation (called "xos_uapi_cipher_sync").
Annexes for example

II.4.2. Differences between tango2 and tango3

Configuration structures and functionalities

The configuration structures used in tango2 and tango3 are slightly different (names of the fields in the structure), so this part of the code has to be ported.

Also, the cipher services have more options on tango3. For example, you can choose the block size (128, 192, or 256 bits) when using AES cipher, while on tango2, only a block size of 128 bits is supported (actual AES cipher).
For this problem, my implementation was to support the special features of tango3 anyway, so that you can use the library for cipher services, even when you don't need the xtask to work on tango2.

DMA cipher engine on tango3

However, the main difference is that on tango2 boards, cipher operations are fully handled by the XPU, while on tango3 boards, a dedicated Direct Memory Access (DMA) cipher engine manages the cipher services. This DMA can read by itself one source of data, make it go through a hardware cipher (AES, DES, RC4…) and store the result to one destination.

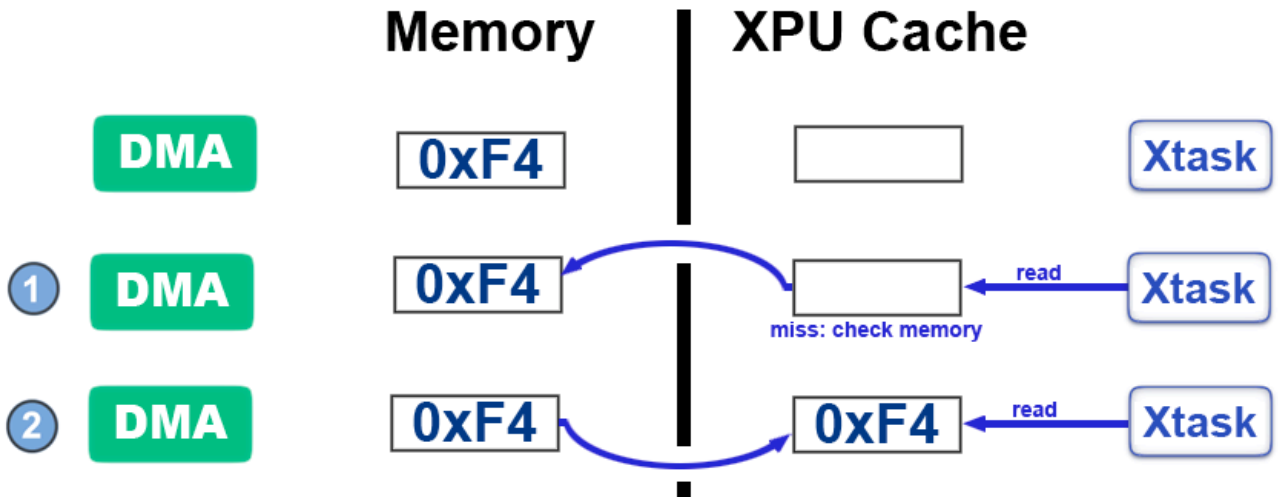
The introduction of the DMA cipher engine on tango3 brings faster encryptions and xtask executions, because the XPU is no longer needed for any cipher service, and so, is free to run anything else while an encryption is going on. On tango2 the XPU is under a heavy workload during an encryption.
Because of this difference, a few adjustments to the code are required in tango3.

Addresses manipulated in a xtask are, in general, virtual addresses (for example, the address of a variable). The DMA doesn't have access to the xtask stack and data, which means it cannot read or write to virtual addresses. Therefore, virtual addresses have to be translated to physical addresses, so that the DMA engine can have access to them.
Also, the DMA can only process data stored in the memory, not in the xtask stack section.

Cache coherency issues on tango3

Also because of the DMA, cache coherence issues have to be handled in tango3. In order to understand this problem, here are a few explanations on how the cache of the XPU works on tango3 boards.

Figure 1: Initial State



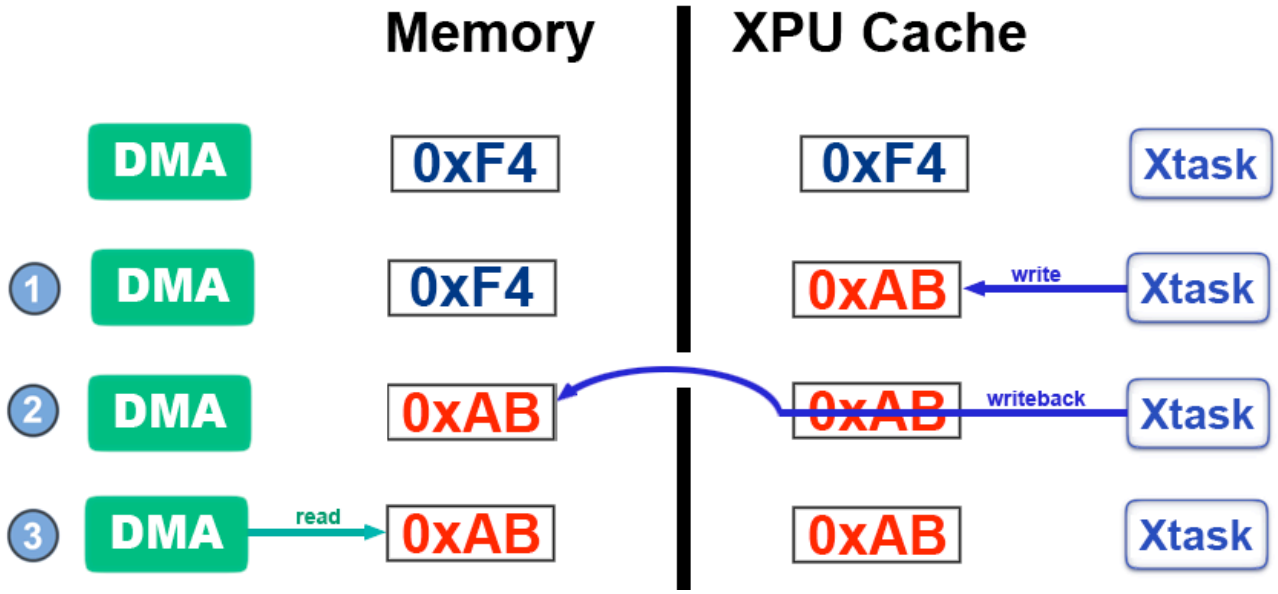
This example is with a random 8 bytes variable, initially stored in the memory, with the value F4. The xtask has never read it or used it in any way. So, the cache of the XPU doesn't contain a copy of this variable yet.

As shown in the figure, the DMA only has access to the memory, and cannot read anything stored in the cache.
The xtask runs on the XPU, and has access to both the memory and the cache of the XPU. However when trying to read data, the xtask will always first try to find it in the cache, and then, if it fails, in the memory.

1. As soon as the xtask tries to read this variable, it first checks the cache of the XPU to see if the variable is available there. As it's not the case, the xtask doesn't find this variable. The failure to find a given variable in the cache is called a cache miss.
As the variable is not in the cache of the XPU, the xtask then tries to read it directly from the memory, which is slower than reading data from the cache.

2. While doing this, it keeps a copy of this variable in the cache, in order to provide faster access to this variable the next time it has to be read or modified. From that point, any change done by the xtask to this variable will only affect the data stored in the cache.

Figure 2: Writeback Operation



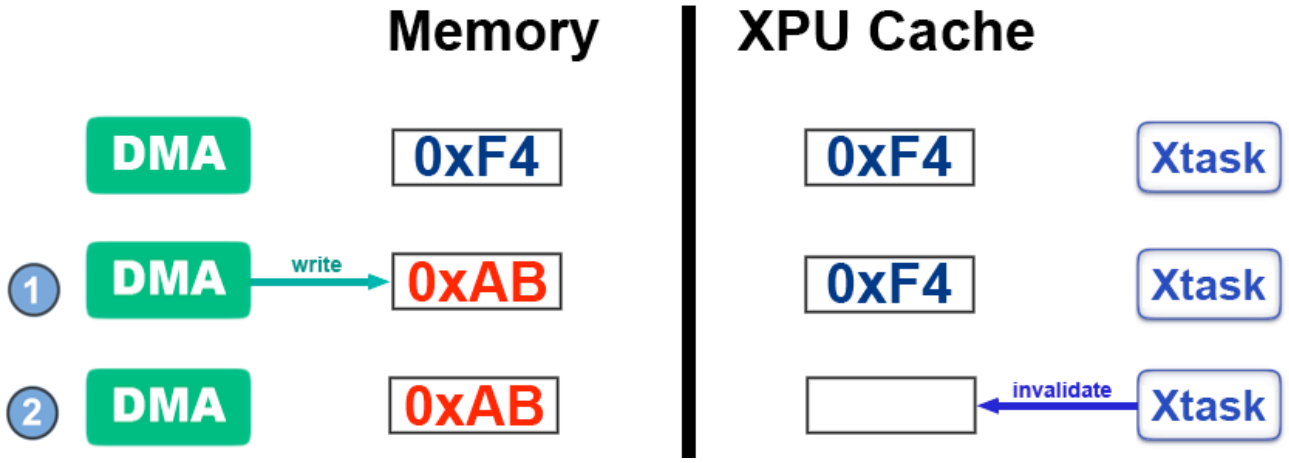
In the initial state, the variable has already been read by the xtask. It's stored in the memory and in the cache.

1. A problem can arise if the xtask changes the value of the variable (going from F4 to AB in the figure above). As the modification is done by the task, it only affects the data stored in the cache of the XPU.
At that point, for this particular variable, there is a different value stored in the memory (what the dma sees) and in the cache of the XPU (what the xtask sees): the cache is not coherent.

2. In order to copy the new value to the memory, so that the DMA can process it (start an encryption for example), a specific instruction has to be executed.
It's called a writeback operation.

3. After this writeback operation, the cache is coherent again: values of the variable in the memory and in the cache are the same.

Figure 3: Invalidate Operation



1. If the DMA changes the value of the variable (as the result of an encryption for example), it will only affect the value stored in the memory. The xtask will still use the value stored in the cache, which is now obsolete. At that time, the cache is no longer coherent.

2. In order to be able to use the new value in the xtask, an invalidate operation has to be done on this variable. This invalidation will delete the copy of the variable stored in the cache.

As a result, the next time the xtask will try to read this variable, it won't find it in the cache and will consequently read it directly in the memory (as seen in figure 1).

To sum up the cache coherency issue:

Before triggering the cipher operation, a write-back operation has to be done on the source address, in order to write any modification done by the xtask to the data stored in the cache, back to the memory.

After the cipher operation, if the xtask needs to access data written by the DMA, the cache of the destination address has to be invalidated to make sure that it matches the new data stored in the memory, written by the DMA.

II.5. Results

The idea behind my library was to create an API that would keep the mode of operation of cipher services on tango2 and tango3: first a configuration structure to fill, and then a function to call with the structure as an argument.

The configuration structure I decided to use in the library is almost the same as the one used in tango3.

The library I wrote is called portlib, and is composed of three separate files:

portlib.h
Header file that defines the API of the library: what the library offers when used in a xtask. It contains the definitions of the portlib configuration structure and the portlib trigger function.

portlib_tango2.c
Implementation of the trigger function for tango2.

portlib_tango3.c
Implementation of the trigger function for tango3.

The two other files are the implementation of the portlib trigger function for tango2 and tango3.

In a xtask using the library, the portlib configuration structure has first to be filled with the right information concerning the cipher operation that has to be done, regardless of the type of the board (tango2 or tango3). Then, this structure is given as an argument to the portlib trigger function.

What happens then in this trigger function, is that it first read the portlib configuration structure, that contains all the information and translates it to the tango2 or tango3 configuration structure by copying and adapting fields in the right way.

Finally, it triggers the cipher operation using the right trigger function (depending on the type of the board), and gives the specific configuration structure that was just created as the argument.
The command that is sent to the XPU at that time is a “classic” cipher command, and is different whether it’s on a tango2 or a tango3 board.
However, the code in the xtask doesn’t change between tango2 and tango3, because it uses the portlib configuration structure and function.

On the tango3 implementation, the translation of the source and destination addresses, and the cache-coherence operations are also handled in the library, so that a programmer doesn’t have to add lines specific to tango3.

III. Project 2
Implementation of a new DRM specification

III.1. Background

Originally called CH-DVD, the CB-HD (China Blue High-Definition) is a new high definition optical disc standard, based on the HD-DVD format, but with additional Chinese-owned technology. This additional technology takes the form of advanced copy protection, designed to prevent the spread of pirated content.

A prototype was demonstrated in 2007, and in early March 2009, Warner Bros announced that they would be supporting the CBHD format, launching with titles including the Harry Potter series and Blood Diamond. CBHD is designed to be an alternative to Blu-Ray, and its main appeal is that discs and players are cheaper to make, compared to Blu-Ray.

To protect the content of CBHD discs, the Advanced Access Content System (AACS) was chosen. It is a standard for content distribution and digital right management, intended to restrict access to and copying of the next generation of optical discs and DVDs.
This standard has also been adopted as the DRM system to protect HD-DVD and Blu-ray Discs. However, AACS for CBHD is a slightly modified version of the standard, designed to provide an even better protection against piracy.

III.2. Project goals

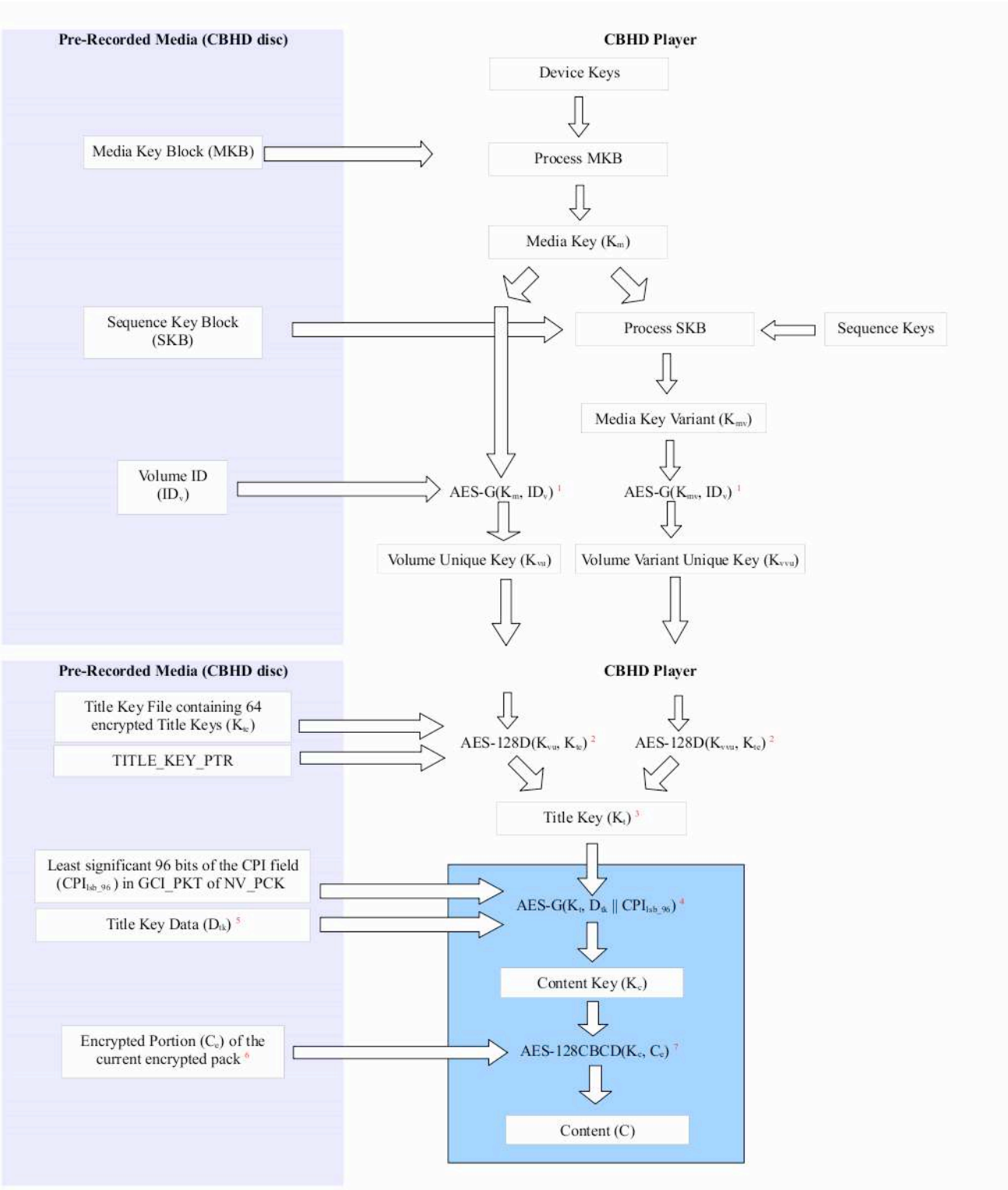
III.2.1. Identification of the problem

A company that sells electronic appliances like TVs, or DVD-players wanted to have CBHD player prototypes ready, in order to present them at a conference about the new CBHD format.
This company uses Sigma boards in the players it sells, so in order to be able to manufacture CBHD players, the CBHD format had to be implemented on Sigma boards, especially the associated DRM system which is AACS. Then, Sigma boards would be able to play CBHD discs.

This company already had implemented by itself the whole AACS decryption process on the CPU of the board, but they found out that the software AES they were using was not fast enough to decrypt the video stream contained in a disc. In that case, decryption is not fast enough when the player is not able to play the film, for example on a TV, with an output of 24 frames per second. If it's slower, the movie is not playing at the right speed, and viewers would immediately notice it.

One way to improve the speed of the decryption was to implement some steps of the decryption process on the XPU instead, in order to make use of the hardware cipher available on the board.

Figure 5: Advanced Access Content System for the CBHD format



This figure shows the whole AACS process required to decrypt a CBHD disc.
All the steps are not important, it just shows that everything derives from the Device Keys, which are the secret keys stored on the player. With those keys, and information stored on the CBHD disc, it's possible to eventually get the "Content" (the decrypted movie).

The blue part is the final step of it, the AES decryption of encrypted packets. One “Encrypted Portion” is 2 kB packet of the film, encrypted using AES cipher. Result of the last step is the “Content”, which is 2kB of decrypted movie. This part is the one that requires the most processing power, and has to be fast in order to handle all the incoming video packets.

III.2.2. Implementing a faster decryption

As explained before, the company had implemented the whole AACS process on the CPU of the Tango2/Tango3 boards (software AES). It wasn't fast enough, and they wanted to use hardware AES instead, by exploiting the XPU. Using the XPU is the only way to get access to the hardware cipher services available on the board (like DES, AES, RSA...).

In order to improve the speed of the decryption, the company asked Sigma to develop the critical part of the AACS process, which is the AES decryption of the continuous incoming video stream (blue part on figure X), with the use of faster, hardware AES instead.

The goal of this project was to develop a package for Tango2 and Tango3 that would allow the decryption of incoming packets using hardware AES.

This package would consist of:

an API running on the CPU, that takes the arguments required to decrypt a packet, and sends them to the xtask.

a xtask running on the XPU, that uses the arguments provided by the API to trigger a hardware AES decryption of the packet.

a test application that shows how the API works.

III.3. Work accomplished

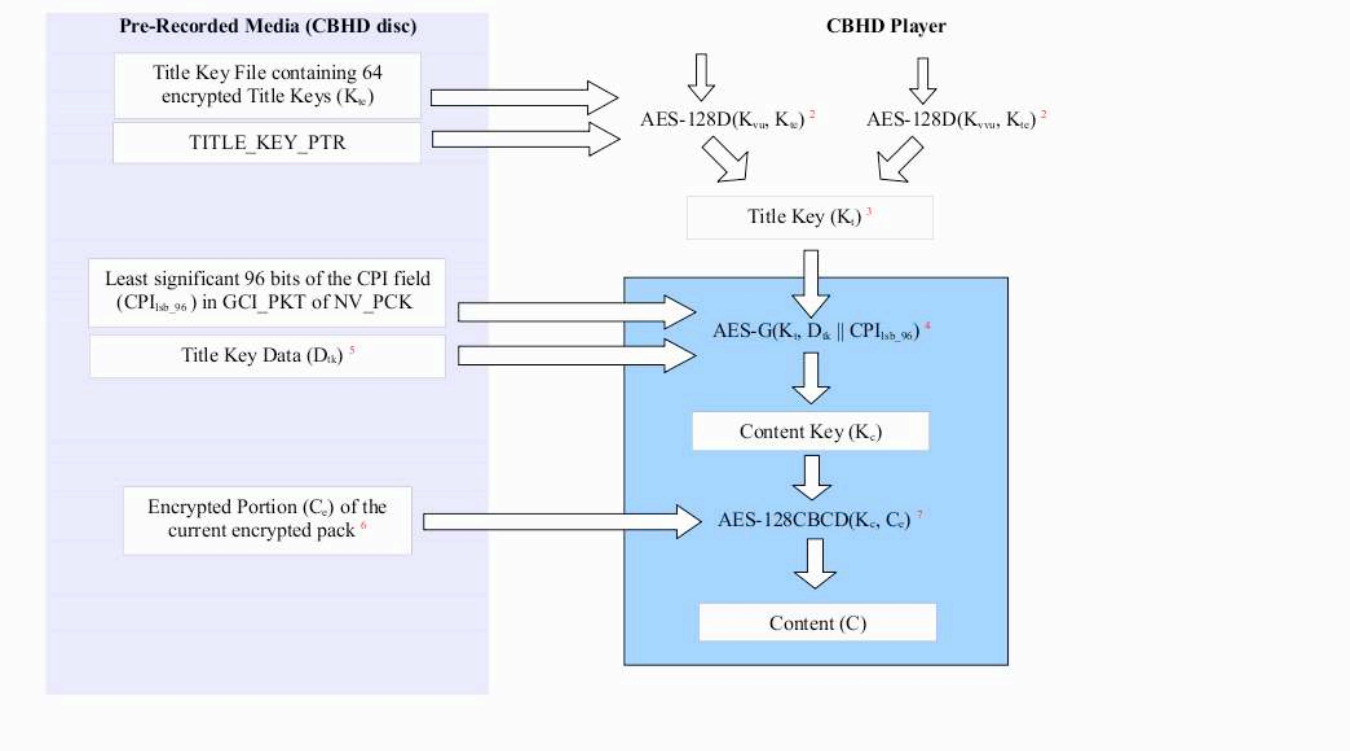
This project was the first DRM implementation I had to write.
The two major parts of the project were programming the xtask, and writing the API.

III.3.1. Writing the xtask

The xtask had to support Tango2 and Tango3. For that reason I decided to use the portlib I wrote during my first project, so that I wouldn't have to port the code that handles the cipher operations (the main part of the xtask).

The two cipher operations that I had to implement in the xtask are in the blue part of the figure:

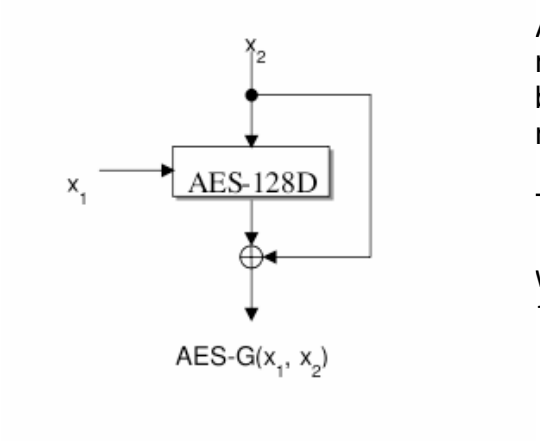
Figure 2: The two steps to implement



The AES cipher is used in the last two steps of the process to decrypt a 2 kB video packet.

First step: AES-G(Kt, Dtk ll CPI_lsb96)

First the Content Key Kc is generated using the Title Key Kt, the Title Key Data Dtk, and the least significant 96 bits of the CPI field. Extracting those two parameters from the CBHD disc, and generating the Title Key had already been implemented.



AES-G is the AES-based one-way function. It is represented by AES-G(x1, x2), where x1 and x2 are 128-bit input values, and AES-G(x1, x2) returns the 128-bit result.

The result is calculated as

AES-G(x1, x2) = AES-128D(x1, x2) XOR x2

where AES-128D(k, d) is the AES ECB decryption of the 128-bit data d, using the 128-bit key k.

Second step: AES128CBCD(Kc, Ce)

The Content Key Kc is then used to decrypt video packets through the AES128CBCD function, which is decryption using the AES algorithm in Chain Block Cipher (CBC) mode.

The Initialization Vector used at the beginning of the CBC encryption or decryption chain is a constant, iv0:

0BA0F8DDFEA61FB3D8DF9F566A050F7816

The result is a decrypted video packet.

Implementing those two functions was the main work on the xtask. Also, in order to handle communications between the xtask and the CPU API, I used an existing library called rpc2xtask, that allows a CPU application to start/stop a given xtask, and communicate with it. Using this library requires the xtask to have a specific structure: a main loop waiting for instructions coming from the CPU application.

III.3.2. Writing the API

At that point of my internship, I only had worked on xtasks (programs running on the XPU), not on CPU applications. The difference is that the CPU applications have to handle interactions with the outside world and interactions with xtasks: starting, communicating, and stopping a xtask.

Most of the problems I encountered while programming this API were due to the lack of documentation: how to compile a CPU application (which headers to include for what functions, what to put in the makefile), how to use the rpc2xtask library. Because there was no documentation, I had to copy and understand the way existing applications were written.
But doing so also creates problems: some parts that are not needed are copied too.

Rpc implementation

The rpc2xtask library acts as a simple communication interface between the CPU and the XPU. After an initialization process where the CPU application starts the xtask, a synchronous call (“rpc call”) can be done from the CPU application to send data / commands to this xtask.

For each packet, one rpc call is done: a configuration structure (made of the address of a buffer containing the encrypted packet, the title key, and the least significants 96 bits of the CPI field) is sent to the xtask. The xtask decrypts the packet by calling the hardware AES service, and sends back an OK or ERROR status to the CPU application.

This first implementation was faster than the initial software AES, but not fast enough for the incoming video stream. The throughput needed to decrypt a CBHD video is 36 Mbps and the RPC implementation only offered a 18Mbps max throughput.

This max throughput is due to the fact that one rpc call is done for each 2048 bytes encrypted packet. As it's a synchronous call, the CPU application waits each time for the return value coming from the xtask (OK or ERROR), but it's actually not used, except if an error happened in the xtask.

A new, faster implementation had to be done, and I was suggested to use a FIFO instead of rpc calls.

FIFO implementation

Instead of relying on slow, synchronous calls, I used a FIFO shared by the CPU application and the xtask. This implementation still relies on rpc2xtask for the initialization step (starting the xtask with the right parameters).

The application fills the FIFO with configuration structures, while the xtask extracts each structure and decrypts the associated packet.
In order to know when decryption is done, I also wrote a function that reads the FIFO and returns the number of structures left in it, zero meaning no more packets to decrypt.

The decryption with this implementation was much faster: the achieved throughput was 45 Mbs, which is enough for CBHD decryption.

III.4. Results

The company was satisfied with the final package, as they stated it was “very good at readability and performance”.

On a side-note, this API for CBHD decryption doesn't use the security features of the Sigma boards.

All the process happening prior to the AES decryption is done in the CPU, but it actually should happen in the XPU only, because secret keys are being manipulated (the Device Keys).
Ideally, the Device Keys should be stored in the serial flash of the board, and a xtask should be able to read it and handle all the operations leading to the decryption of the CBHD disc. The only data transmitted from the CPU to the XPU should be information (Volume ID, Media Key Block, encrypted content, etc...) coming from the CBHD disc (that is not secret).

The reason why it wasn’t implemented in a secure way was that the company only needed a working prototype in order to present the CBHD format at a conference.

IV. Project 3
CPU-XPU Communication performance measure

IV.1. Background

A customer wanted to implement, on the tango2 and tango3 boards, a program that interacts with an internet website. However, the website might be not available at a certain time, or the board might not have access to the internet. Hence a timeout value has to be implemented, so that when the board tries to connect to the website, if it takes too much time, it just considers the website not accessible and returns an error.

In order to know what the value of this timeout should be, the customer needed to know how much time is spent in the board when it tries to connect to the website, going from the XPU (where the call will come from) to the network output of the board.

The program was using the rpc2xtask library (introduced in project 2) to handle communications between the CPU and the XPU.

So, the goal of my project was to measure the time it takes to communicate between the CPU and the XPU, with various applications running at the same time on the board.

I had to measure the time a single synchronous rpc call takes when:

- No other programs are running on the board
- A simple application displaying color bars on the TV is running
- A video is being played on the TV at the same time
- An AES encryption is going on at the same time

The measures had to be done on tango2 and tango3.

IV.2. Work accomplished

I first carried the work on a tango2 board. I had to write two programs:

The main application running on the CPU, that includes the rpc2xtask library, and triggers rpc calls to the XPU. It also measures the time it takes to get the return value from the XPU, after the rpc call.

The xtask running on the XPU, that also implements the rpc2xtask library, and whose function is to just answers to the calls coming from the main application.

As a classic problem when doing performance tests, I had to make sure that the measures I was doing were not interfering with the normal behavior of the program, ie were not changing the duration of the calls.

IV.2.1. Linux Time command

In order to get a general idea of the average time a rpc call takes, I first used the “time” Linux command. This command takes a program as an argument, and returns the time it took for this program to run.

The CPU program I wrote was making 10 000 rpc calls. However the duration measured by the time command is for the whole program. So, it takes into account the 10 000 rpc calls, but also the initialization of the program, and any other operation happening in the program, not related to rpc calls (loops, conditional statements, etc…), that should not be measured.

Still, it gave me an idea of the average duration of a single rpc call.

IV.2.2. Register reading

To get more precise values, I had to make measures inside the program, in order to directly measure the time taken by each rpc call statement.

One way to handle time on the tango2/tango3 boards is to use a specific register, accessible at a certain address. This register is updated with a 27 MHz frequency and it’s possible to read its value from a cpu application. I used this 27 MHZ counter to measure durations in my program.

The program has two parts (see anexxes XXXXX). First, it measures the time taken to make 10 000 rpc calls, in order to get an average value for one rpc call.

Then it measures one by one, 10 000 other rpc calls, and returns the longest, and the fastest call of this run.

Finally the program returns the three stored values: average (from part 1), and longest and fastest (from part 2) rpc call.

One thing that I kept in mind was that I was doing the measures in the same environment (the CPU program) as the values I was measuring. Hence, I was afraid that they would interfere with the program and the rpc calls. As shown in the annexes XXXXX I tried to make sure that the measures were separated from the rpc calls.

Here is the description of the CPU program, in pseudo-code:

Initialization

```
Initialization of the program, the xtask, and the rpc2xtask library
Synchronization with the xtask using rpc2xtask
```

Part 1: Calculating the average duration

```
Time1 = Get time
Make 10 000 RPC Calls
Time2 = Get time
Average duration = (Time2 – Time1) / 10 000
```

Part 2: Calculating shortest and longest durations

```
Do 10 000 times:
     Time1 = Get Time
     Make 1 RPC Call
     Time2 = Get Time
     Duration = Time2 – Time1
     Store Duration if it's the longest so far
     If not, store Duration if it's the shortest so far
End loop
```

Final Part

```
Print Average duration
Print Shortest duration
Print Longest duration
```

Here is the output of the CPU program in my linux shell:

Initialization

```
Starting Xtask
xtask started with handle 0x00003400
Xtask started !
```

Rpc2xtask synchronization between program and xtask

```
Detecting Xtask... OK
```

For each RPC call

```
Xtask RPC Call
Sent signal to xtask
Rcvd signal from xtask.
```

End of RPC call

End of the program

```
Results
Highest 4604 us
Lowest 2653 us
Average 3918 us

Xtask terminated !
```

IV.3. Results on tango2

On tango2, the results were good, in the sense that all the values were close to the average, and nothing unexpected happened. The more the cpu and the xpu were busy (another program running at the same time for example), the more time the rpc calls were taking. Hence, the measures seemed right, and accurate. The results for tango2 are available in the annexes.

The next step was to port the program and the xtask on tango3, which required very little work as the rpc2xtask library was already taking care of most of the tricky parts in the porting. The program and the xtask themselves are almost identical on tango2 and tango3.

IV.4. Results on tango3

On tango3, the results were much more surprising, because I was getting a few very long calls (100 times longer than the average) during the then thousand rpc calls of the part 2.

I tried to track and find what was causing this “freeze”, by measuring the time taken by separate instructions in my program, and in the rpc2xask library.

I found that sometimes, for no apparent reasons, some operations happening in the rpc2xtask library when making a rpc call, were taking a lot more time than usual. I also found out that in multiple runs of the program, it never was the same line of code that was “freezing” the program.

For this reason, it was very difficult to understand what was happening, as it could have been caused by many things: cpu interrupt/scheduling, code optimizations done at the compilation, access time to resources like memory.

I couldn’t get a better understanding of what was happening unless I looked at the source code of the operating system, xos. It was decided that testing and modifying xos was too complicated for me at that time, and as only the actual time measures were needed for the customers, the project was finished.

However I realized that the output of the cpu program was reporting a lot of information, especially every rpc calls with the following lines:

```
Xtask RPC Call
Sent signal to xtask
Rcvd signal from xtask.
```

My program was doing 10 000 rpc calls in part 1, and 10 000 other rpc calls in part 2, so those three lines were printed 20 000 times on the screen, at a very fast rate (an average rpc call takes 60 us) and the output was overflowing my linux shell.

I tried to run the program with a redirection of the output to a file instead of the linux shell, and the times of the rpc calls (average, longest and shortest) became significantly smaller, but there still were a few very high values.

Finally, I removed the entire log and debug commands in the rpc2xtask library, and ran the program again, which, consequently, gave the following output:

Initialization

```
Starting Xtask
xtask started with handle 0x00003400
Xtask started !
```

Rpc2xtask synchronization between program and xtask

```
Detecting Xtask... OK

Running tests…
```

End of the program

```
Results
Highest 4007 us
Lowest 2157 us
Average 3462 us

Xtask terminated !
```

This time, the durations were even smaller, but most of all, there wasn't any very high values anymore. So this problem was probably caused by an overflow of the standard output, because the program had to print on the screen too much data at the same time. This caused the "freezing" of the program that was happening randomly during a few rpc calls.
The values that I got after this modification are in the annexes, and were the one I sent to the customer.

I also modified the tango2 version of the program in order to remove the output of the rpc2xtask library too, and the values I got were also a little smaller.

V. Project 4
Serial Flash access library

V.1. Background

The serial flash is the third part of the Secure Media Processor architecture. It's a secure memory designed to store reboot-persistent, secret data, that is going to be used by the XPU to decrypt content.

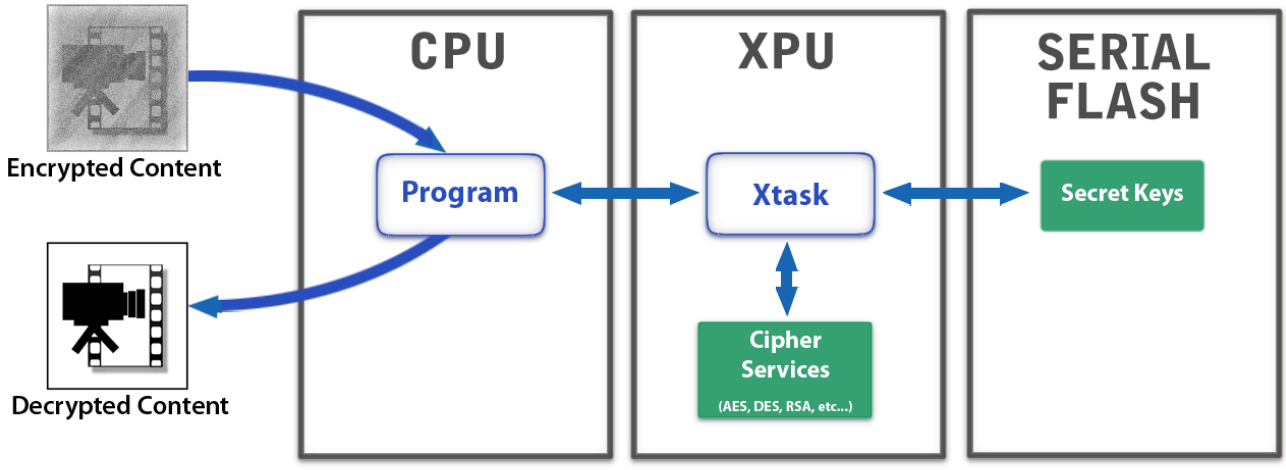
This memory is encrypted, and can only be read by the XPU. In general, the serial flash is used to store secret keys required by DRM applications.

The secret keys are first written during production. Then, when the board is sold to a customer, in a DVD player for example, it has the required secret keys already written in the serial flash, ready to be used to decrypt DVDs.

It is very hard to hack into the serial flash itself, and the only way to read that memory is to do it from the XPU, which is also secure. Secrets stored in the serial flash should only be used by the XPU, and should never appear on the CPU side of the architecture (like in a program).

For example, in the AACS specification, intended to restrict access to and copying of Blue-Ray discs, the decryption process uses a "Device Key" that is specific to each players.

Hackers have been able to find such keys by using debuggers to inspect the memory space of Blu-ray player programs that were weakly protected. The keys were then released over the Internet, and the first unprotected HD movies became available. This explains why it's all-important to protect secret keys of DRM standards, and that's the purpose of the serial flash.



The goal of project was to write a library that allows a user to write and read anything in the serial flash, using a CPU application. It was requested by a customer of Sigma, because he wanted to make tests with the serial flash on the boards.

V.2. Work Accomplished

In order to be able to write such library, I first had to understand how the serial flash works. It was the first part of my work on this project.

V.2.1. Analysis of the Serial Flash

I looked at existing xtasks that interact with the serial flash, in order to figure out how it works. The way the serial flash works is closely related to the way xtasks are designed. The xtask signing process is the first thing to understand.

Xtask signing

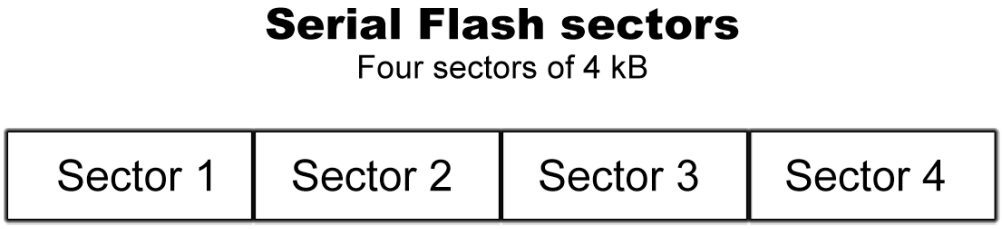
Every xtask is signed by Sigma Designs, so that it's not possible to write and run a custom xtask, that has not been checked by Sigma. Sigma has created many different certificates, to sign xtasks of different functions. Sigma uses a little more than 10 different certificates for all the xtasks.

When a xtask is loaded on a board, the os checks that it has Sigma's signature. If it's not the case, the xtask is not allowed to run.

The certificate the xtask was signed with, is also used when interacting with the serial flash.

Serial flash characteristics

The serial flash is an encrypted memory, divided into four sectors of 4 kB. Each sector is initially empty and has no owner.



The only way to interact with a sector is to use the serial flash API available in xtasks.

Xenv API

Xenv stands for “secure environment”, which is the serial flash.

Each sector of the serial flash can only be accessed from a xtask, using the following commands:

Xenv_chown (sector, certificate_sha1)

Chown stands for change_owner. There are two cases when the xenv_chown function has to be used:

When a xtask wants to write data to a sector that has never been used before. It first has to get the ownership of that sector. Initially, a sector is not owned, so a xtask can claim the ownership by using a xenv_chown command, with sector number (1, 2, 3 or 4) as the first argument, and the SHA-1 of its own certificate as the second argument.

If it succeeds, the sector is then owned by that certificate. The xtask can now read and write freely to that sector, and so can every xtask that has been signed with that specific certificate.

When a xtask wants to give the ownership of a sector it owns, to another certificate. It uses the xenv_chown command with the SHA-1 of the new certificate as the second argument. If it succeeds, the sector is then owned by the new certificate, and the xtask that used the xenv_chown command can no longer read or write to that sector.

Xenv_format(sector)

Format the given sector on the serial flash, erasing any data stored. This will succeed only if the sector is owned by the certificate of the xtask calling the command.

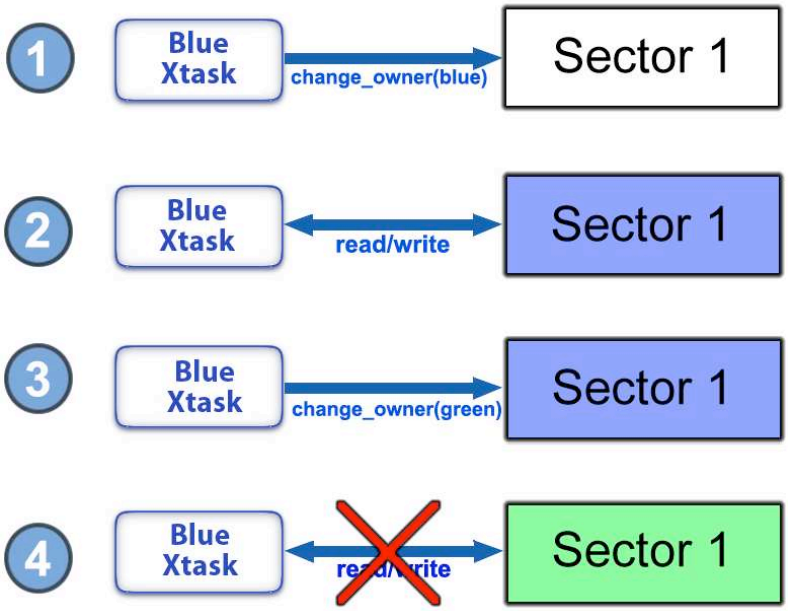
Xenv_get(sector, …) and **Xenv_set(sector, …)**

Those functions are used to read (xenv_get) and write (xenv_set) data to the given sector of the serial flash. I haven’t specified the arguments of those functions because it’s not relevant.

They will succeed only if the sector is owned by the certificate of the xtask calling the command.

Figure 3: Sector ownership sum up

Figure 1



1. Initial state: the sector is not owned by any certificate. The xtask running on the XPU was signed with the blue certificate.
The xtask uses the change_owner command with its own certificate, the blue certificate, as the argument.

2. The sector becomes owned by the xtask's certificate. The xtask can now write and read data freely in this sector.

3. As the sector is owned by the xtask's certificate, the xtask can use the change_owner command to give the ownership to another certificate.

4. The sector is now owned by the green certificate. As the xtask was not signed with this certificate, it can no longer read or write any data to the sector.
Also, the only way to change the ownership of this sector now is to run a new xtask, signed with the green certificate, that uses the change_owner command.

V.2.2. Writing the library

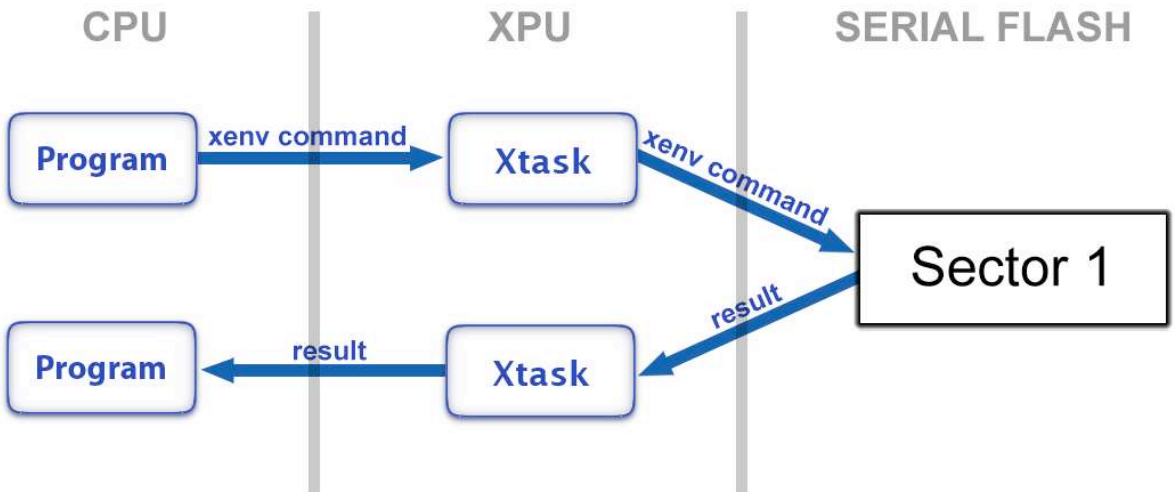
After the analysis part, I started writing the library. I decided to call it “xenv_access”. The purpose of xenv_access is to give access to the serial flash from a CPU application. In other words, I had to make the xenv API, initially available only in xtasks, accessible from a CPU application also.

This meant that I would have to write two separate things:

A library for CPU applications, that defines the xenv_chown, xenv_format, xenv_get and xenv_set functions, and send the commands and the arguments to the xtask.

A xtask acting as an intermediary between the CPU program and the serial flash. Its only function is to read xenv commands coming from the CPU through the library, translate them to actual serial flash commands, and send back the result to the CPU application.

Figure 2: Xenv Access



I used the rpc2xtask library (introduced in project 2) to handle communications between the CPU program and the xtask.

Writing the API and the xtask was a similar task to my previous projects.

V.3. Problems Encountered

V.3.1. Security threat

One problem is that this library could threaten the security of the whole architecture, because it allows any program to read and write freely to the serial flash. This is usually not allowed because, as the XPU and the serial flash are supposed to be secure, the CPU is not, and can be “easily” hacked.

To prevent any misuse of such library (like reading secret keys previously stored in the serial flash), a special certificate was created for the xtask that is part of this library. As the certificate will only be used to sign this specific xtask, it will never be possible to read secret data that was previously written by another xtask, because the sector of the serial flash will be owned by a different certificate.

V.3.2. Sectors destruction

Because of the way the serial flash works, a wrong operation can completely lock a sector.

This happens if a xtask uses the change_owner command to give the ownership of a sector to another certificate. It gives the SHA-1 of the new certificate as the argument of the xenv_chown command, and if it succeeds, the sector is then owned by that new certificate.

However, if the SHA-1 that is given, is, for any reason, wrong, the sector gets a new ownership that is not related to any certificate used by Sigma. Then, the sector is completely locked. The only way to unlock is to write a xtask that changes back the ownership, and sign it with the certificate that owns the sector. However, if the SHA-1 was wrong, there is no such certificate, so, there is no way to unlock the sector.

On development boards, used by engineers to test program and xtasks, it’s still possible to reset the entire serial flash. But, on production boards, the sector is lost forever. Basically, if a program is used during the production chain of boards, to write keys in the serial flash, and puts wrong ownership on the sectors, every board looses the sectors.

In my library, I made a mistake in the code of the xenv_chown function. The first argument is the sector number, and the second argument is the SHA-1 of the certificate that is going to get the ownership of that sector.

There is a step where I copy the SHA-1 that was given by the CPU program, to a buffer, in order to send it to the xtask. However during this copy, I was incrementing the SHA-1 by two as a mistake. So, every time a program was using the chown command through my library, it would send a wrong SHA-1 to the xtask, and the xtask would put the same wrong SHA-1 as the new owner of the sector, completely locking it.

I sent a version of the library, with this error, to the customer. This could have been a disaster. Hopefully he only tried it on one board, saw that something was wrong, and came back to me. Then I corrected the code.

VI. Project 5
Key-Loading Tools

VI.1. Background

VI.1.1. Purpose of the key-loading tools

As seen in the previous project, the serial flash is a secure memory, whose purpose is to store secret keys that are needed by DRM applications. The usual mode of operation is to first write the keys that are going to be needed later, during the production of boards. Later, when the board is sold, as part of a DVD player for example, it has the keys already stored in its serial flash, so that the board is able to use them in order to decrypt DVDs.

In order to be able to write keys to the serial flash, a set of tools was created: the key-loading tools. They consist of a loader application that runs on the CPU of the board, and, for each DRM standard (AACS for blue-ray, JANUS for Microsoft content, etc..) one xtask, that runs on the XPU, and that can write and read directly in the secure environment. Each key-loading xtask is written for a specific DRM standard.

The customer uses the CPU application to send commands and data to the xtask. Depending on the DRM specification it was designed for, the xtask will write the data (secret keys) to the right location in the serial flash.

These tools are used by customers to write keys on the boards in the production chain. As soon as the keys are written, the board can use them anytime, when incoming video content has to be decrypted. Needless to say, the keyloading tools have to be extremely reliable, as a writing error during production could compromise many boards (as seen in project 4).

VI.1.2. Goal of the project

Sigma boards support around ten different DRM specifications (AACS for blue-ray, JANUS for Microsoft content, etc..). Each specification has its own key-loading xtask, whose only function is to write and read keys of the specification to the serial flash.

The key-loading tools are also composed of one loader application running on the CPU, that is able to interact with any of the key-loading xtasks (they share the same communication interface). Its purpose is to send data and commands to the xtask.

Within the ten specifications, half of them require only one key, of a specific size, to be written in the serial flash. Therefore, the key-loading xtasks of those “simple” specifications perform very similar operations, the only difference being the size of the key to write. Other “complex” specifications require more than one keys to be written in the serial flash.

Changing the general behavior of the keyloading tools (adding a new type of operation for example) would require implementing the change in every xtask.

The goal of project was to re-write the key-loading tools. First, improve the CPU application with new functionalities (more error codes, bugs corrections…), and make it more convenient to use (possibility to use it through a command line).

Then, write one ”universal” key-loading xtask, that handles all the different specifications, so that all the similar functions are merged into one (making the code easier to maintain). For “complex” specifications, write separate, specific functions, but that are still parts of this xtask.

As usual, this work had to be done for tango2 and tango3.

VI.2. Work accomplished

The first part of the work was to study the actual key-loading tools and the DRM specifications supported by Sigma.

VI.2.1. DRM Specifications

I separated the ten DRM specifications supported by Sigma in two different types:

“simple” specifications, where only one key, of a specific size has to be stored in the serial flash. These specifications all use the same functions (check the size of the key, write to the serial flash), the only thing that changes is the size of the key. The key-loading xtasks for those specifications are very similar.

“complex” specifications, where many keys have to be stored, or where special operations have to be performed before writing the key to the serial flash (a decryption for example). These specifications require special code to work, and the key-loading xtasks for those specifications are quite different.

VI.2.2. Mode of operation of the key-loading tools

The user first has to manually load the key-loading xtask that is written for the DRM specification he is working on. While doing so, he has to specify some parameters to the xtask:

the sector of the serial flash to use

the writing attributes of the data that is going to be written (Read only, Read / Write)

a communication address used to communicate with the CPU application.

After starting the xtask, the user now has to start the CPU application that is going to send data and commands to the xtask. At that time, he has to specify other parameters to the application:

the operation to do, it can be either loading a key, or checking if there is a key stored

the path to a file containing the data to write to the serial flash, in case of a load operation

the encryption mode, which is used if the user wants to encrypt the key before writing it to the serial flash

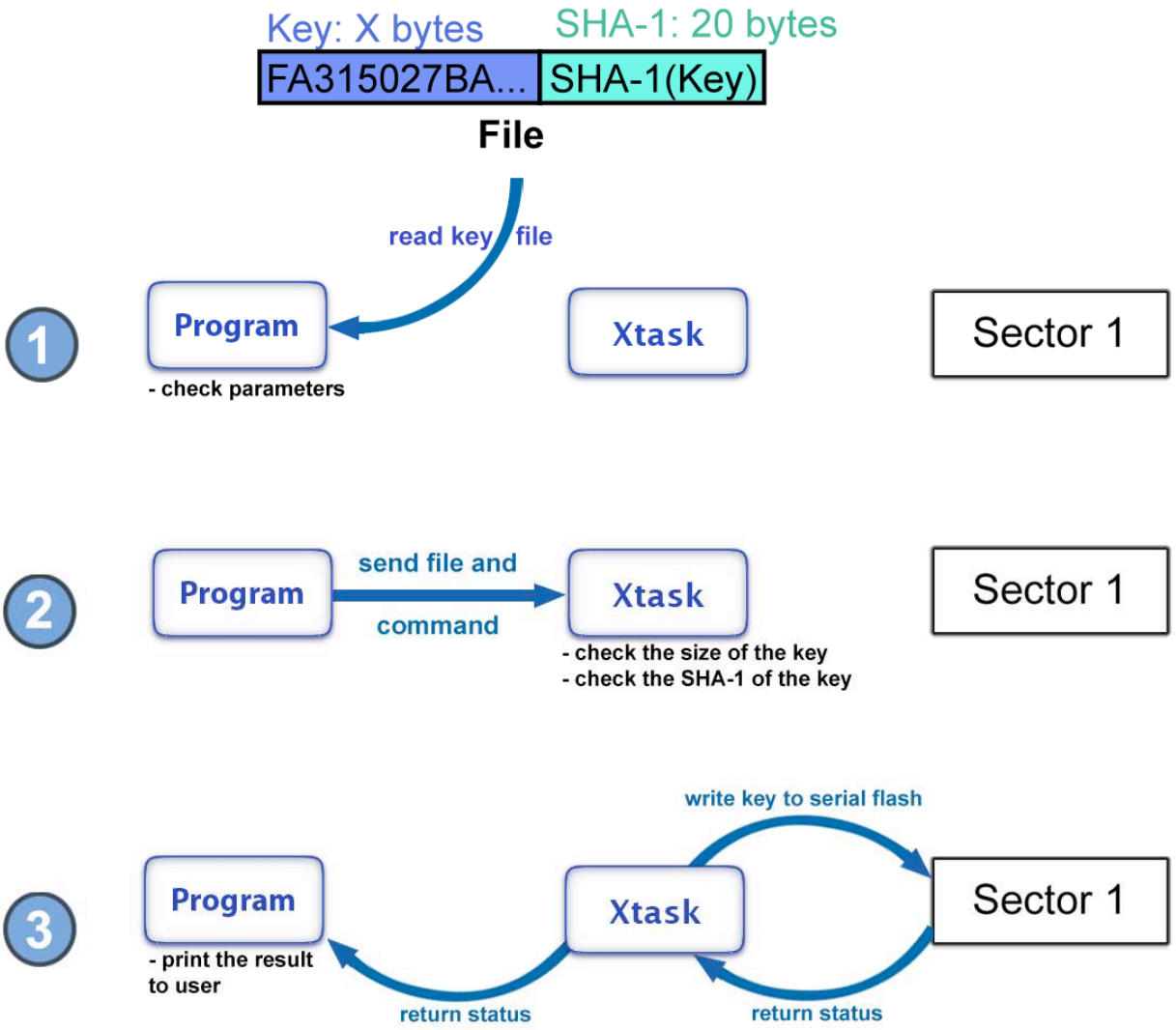
a communication address, that has to be the same as the one used when starting the xtask

When the CPU application is started, it executes the right command, sends data to the xtask, and returns the result of the operation.

As you have to manually start the xtask, and then the program, it is not very convenient to use the tools.

Here is an explanation of what exactly the program and the xtask do when a key has to be loaded.

Figure 2: Loading a key to the serial flash



1. The user starts the CPU program and the key-loading xtask, with the right parameters.
The CPU application first reads the file that contains the key to load.
The file should have the following format: first the key, and then, appended to it, in the same file, the SHA-1 of that key.
The program also checks the parameters it was started with.

2. The program sends to the key-loading xtask the command (in that case: load a key), the parameters, and the file containing the key and its SHA-1. The xtask checks the size of the key (has to be the size expected by the DRM specification) and the SHA-1 that is appended to it.

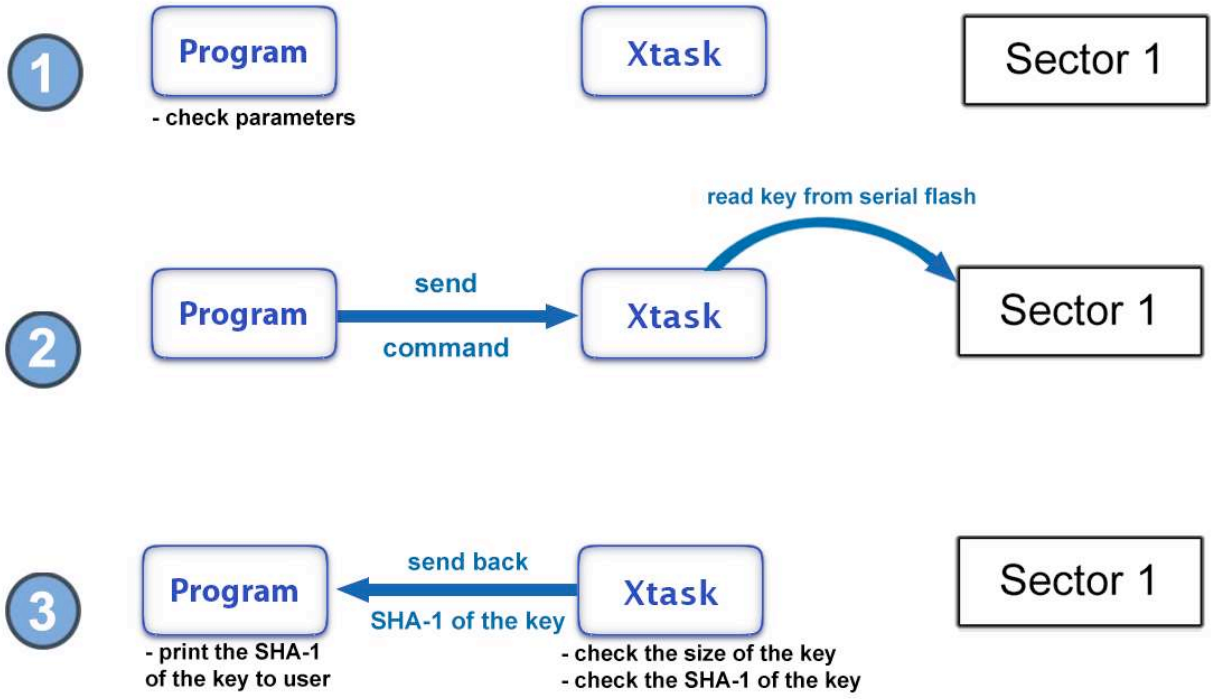
3. The xtask writes the key to the serial flash sector, at the address designed for the DRM specification. Then, it sends back to the program the return status (error or OK). The program prints this status to the user.

Expected output of a load operation

```
DTCP Keyloading tool.
Load operation.
Detecting Xtask... OK
Reading key.bin : 148 bytes.
Appended SHA1 of the record:
f79ca2c3f7d723241d9b7748519e1d739c78a6f0
Sending record to xtask... 148 bytes sent.
Sending load command to xtask... record saved.
Xtask terminated !
```

Here is an explanation of what exactly the program and the xtask do when a key has to be loaded.

Figure 2: Checking if there is a key already stored in the serial flash



1. The user starts the CPU program and the key-loading xtask, with the right parameters. The CPU program checks the parameters it was started with.

2. The program sends to the key-loading xtask the command (in that case: check if there is a key) and the parameters. The xtask tries to read a key in the serial flash using those parameters.
If it finds a key, it checks the size of that key (has to be the size expected by the DRM specification) and generates the SHA-1 of it.

3. The xtask sends back to the program the SHA-1 of the key stored in the serial flash. The program prints it to the user.

Expected output of a check operation

```
DTCP Keyloading tool.
Check operation.
Detecting Xtask... OK
Sending check command to xtask ...Record check OK - SHA1 of the record:
f79ca2c3f7d723241d9b7748519e1d739c78a6f0
Xtask terminated !
```

VI.2.3. Rewriting the CPU program

I used most parts of the present CPU program of the key-loading tools.

What I added was functions to automatically load the xtask, so that the user doesn't have to start the xtask manually. This allows the user to do everything with one command line, the one that starts the program.
Coding this took some time, because it was the first time I was writing functions to automatically load a xtask. I had to look at other programs to see how it can be done. The code that does this is very "close" to the operating system of the board and therefore is hard to read and understand. Also, for the same reason, the code changes a lot between tango2 and tango3 (they use a different operating system).
Eventually, and with some help, I managed to have the program reads the xtask in a file, and load it on the board.

Initially the user had to give some parameters to the xtask when he was starting it: sector of the serial flash to use, a communication address, and the writing attribute in the case of a loading operation. As the xtask is automatically started in the new version, I had to change that, and made it so that those parameters should be given at the start of the CPU program instead.

The last thing I did on the program was to improve the communication interface with the key-loading xtasks, by adding more error codes, so that the output gives a better and a more precise explanation, if an error happens.

While writing this new key-loading program, I also made sure it would still work with the present key-loading xtasks, and also with the new universal xtask I was going to program next.

VI.2.4. Writing the universal xtask

After writing the new key-loading program, I started working on the new “universal” xtask.

In the “old” key-loading tools, the user just loads the right xtask for the DRM specification he uses.
With my new key-loading program, it is started by the user with multiple parameters, including the DRM specification to follow.
As the new xtask designed to support all the DRM specifications, I had to add an extra step to this universal xtask, where it communicates with the program in order to know which specification to follow.

I also improved the way errors were handled so that there are more different error codes, giving a better description of the error.

Finally, I had to make the universal xtask able to load and read keys of the many DRM specifications supported by Sigma.

As stated before, in the case of “simple” specifications, the associated key-loading xtasks perform very similar operations (writing a key of a certain size to the secure environment, extracting a key),
In order to handle “simple” specifications, I just had to copy the functions of one of the “simple” key-loading xtask and adapt them a little, to make them work in the universal xtask.
Then, I made the five “simple” specifications rely on the same functions in the universal xtask, instead of having five different xtasks.

In the case of the “complex” specifications, I had to copy the code of the associated key-loading xtasks and keep the code of each specification, as a part of the universal xtask (no possible simplification).

VI.3. Results

Behavior of the new keyloading tools

The user just starts the CPU application with all the arguments required:

the sector of the serial flash to use

the operation to do, it can be either loading a key, or checking if there is a key stored

the DRM specification to follow

the path to a file containing the data to write to the serial flash, in case of a load operation

the path to the key-loading xtask

the encryption mode, which is used if the user wants to encrypt the key before writing it to the serial flash

the writing attributes of the data that is going to be written (Read only, Read / Write)

Some of those arguments have a default value if not specified in the command line. Here is an example of a command line to run the program:

```
./loader -aacs -sector 1 -cmode 0 -access RW -load key.bin
```

The CPU application will then load the xtask itself, and interact with it in order to execute the right command and return the result of this operation.

This mode of operation is much more convenient because the user doesn't have to first load the xtask. Hence, it's possible to do one operation with only one command line, and for example chain them in a script file.

Output of a load operation

```
DTCP Keyloading tool.
Load operation.
Reading xtask.xload : 21460 bytes.
[xhandleT3.c:169] xtask loaded to image 0x00001200
[xhandleT3.c:111] xtask started with handle 0x00003400
Xtask started !
Detecting Xtask... OK
Reading key.bin : 148 bytes.
Appended SHA1 of the record:
f79ca2c3f7d723241d9b7748519e1d739c78a6f0
Sending record to xtask... 148 bytes sent.
Sending load command to xtask... record saved.
Xtask terminated !
```

`

VI.4. Problems encountered

VI.4.1. Pre-existing bugs

The first part of my work was to look at the present key-loading xtasks of each DRM specification. A problem I had is that some of them still had bugs and mistakes, so it made it harder to understand the code: in some xtasks, SHA-1 checks were not working, or the output was not correct.

AACS specific functions

AACS is a complex specification because it uses many different keys, and the total size of the data that has to be written in the serial flash for AACS fits in 2 sectors of the serial flash (2 * 4kB).

So the xtask has to request the ownership of two sectors instead of one, before writing anything.

When I started to look at the code of the xtasks in order to understand the way they work, I found that many of them were requesting the ownership of two sectors, but then using only one to write and read data.

I found out that It was because AACS was the first key-loading xtask to be written, and as some parts of the key loading xtasks are similar, when a new DRM specification had to be supported, the programmer that wrote the xtask for this new specification just copied some parts of the AACS xtask, but didn't remove what is specific to AACS.

So I found in many key-loading xtasks the double ownership request, when it was only needed for AACS.

VI.4.2. Xos2 sector ownership bug

I was using the xenv_access library (project 4) to debug my new “universal” key-loading xtask: reading the serial flash, in order to check what the universal xtask was actually writing there, and to make sure it was working. While doing this, I got some errors, on tango3 only, making it impossible to read or write to the serial flash.

I found out that there was a problem with the ownership of the serial flash sectors (see project 4 for sector ownership explanation).

First the universal xtask was taking the ownership of the sector, in order to write a key there. Then I was passing the ownership of that sector to the xenv_access xtask in order to use it to read what the universal xtask had just written in the serial flash and make sure it was what was expected.

However the change of ownership was not happening, so that the xenv_access xtask was never able to read the serial flash. What was strange is that I wasn’t getting any error from the operating system when doing the change of ownership, as if it was working, but it wasn’t.

I asked my superior about this issue. As there was no apparent reason for the change owner to not work in my xtasks, he decided to have a look at the code of the operating system, xos2, just to check the functions related to serial flash ownership.

He found out that there was a bug in the way xos2 (the operating system of tango3 boards) was handling it. Changing the ownership would only work on a sector that is “blank” (never been owned by any certificate). Then, changing the ownership a second time would not return any error, but nothing would be done, so that the sector just keeps the same, first ownership, forever.

This bug was quite serious, and meant that a new version of xos2 would have to be released in order to fix it. Then this new version would have to be sent to the customers so that they can update their tango3 boards.

VII. Conclusion

This master thesis gave me the opportunity to understand how security is handled, and how content is protected in video players. As explained during this report, content protection relies on specific hardware and software, and the many projects I worked on gave me a good understanding of each aspect.

More generally, this thesis was about embedded systems, cryptography, C language, and real time systems. This allowed me to use and to broaden the knowledge I acquired at Chalmers during my master program. I learned a lot, because while adding and improving features to Sigma board, I encountered many actual, “classic” issues when dealing with embedded systems (cache coherency, limited memory/processing power, etc…).

However, there is of course still a lot to learn and to do.

As new formats and standards emerge, new, more sophisticated DRM systems are developed in parallel. This requires further work and investigation to implement those new standards.

Yet the future of DRM systems is not clear. Recently, the music industry has decided to stop protecting audio content, because impact on piracy has been limited, and it just punished customers that had sometimes trouble playing the audio content they bought, because of a badly designed DRM system.

However, the film industry is far from getting there yet.

VIII. Appendix

VIII.1. Project 1

VIII.1.1. Classic implementation of an AES Decryption for tango2 and tango3

```
#define AES_BLOCK_SIZE 128
RMuint32 * src,dst,iv,key;          // Pointers to source buffer, destination buffer, key and iv
RMuint32 size;                      // Size to process

// Fill in the src buffer, the iv, and the key.
.......

// If compiling for tango2
#if (EM86XX_CHIP == EM86XX_CHIPID_TANGO2)
// Configuration structure
        union cs_config config;

        config.aes.cmd = aes_cmd_k128_b128_cbc_decrypt ;   // AES cipher, key size 128 bits, blocksize 128
                                                           //bits, CBC decrypt mode

        memcpy(config.aes.iv , iv, 4);                     // Initialization vector
        memcpy(config.aes.key , key, 4);                   // Key
        config.aes.src = src;                              // Source address
        config.aes.dst = dst;                              // Destination address
        config.aes.blockcount = (size/AES_BLOCK_SIZE) – 1; // Number of AES blocks to process

// Trigger AES cipher operation
        sync_cipher(cs_xpu_aes, &config)


// Else if compiling for tango3
#elif (EM86XX_CHIP == EM86XX_CHIPID_TANGO3)

        struct dmacipher_ctx ctx;                          // Configuration structure
        RMuint32 rwga;
        rwga=xos_uapi_getrwga();                           // Needed to make address translations

        ctx.mode = (1<<LOG2_DMACIPHER_MODE_AES);           // AES cipher
        ctx.config.aes.aesmode = AES_CBC_DEC ;             // AES CBC decryption mode
        ctx.config.aes.keysize = AES_BLOCK_SIZE;           // Size of the key: 128 bits
        ctx.config.aes.blocksize = AES_BLOCK_SIZE;         // Blocksize: 128 bits
        memcpy(ctx.config.aes.iv , iv, 4);                 // Initialization vector
        memcpy(ctx.config.aes.key , key, 4);               // Key

        ctx.src = VA2GA(src, rwga);                        // Translation of source address
        ctx.dst = VA2GA(dst, rwga);                        // Translation of destination address
        ctx.size = ((size/AES_BLOCK_SIZE) – 1)*AES_BLOCK_SIZE; // Bytes to process

        xos_uapi_dcache_range_w((src, size);               // Writeback source address
        xos_uapi_dmacipher_sync(&ctx);                     // Trigger cipher operation
        xos_uapi_dcache_range_i(dst, size);                // Invalidate destination address
#endif
```

VIII.1.2. Implementation of an AES Decryption using portlib

```
#include “portlib/cipher.h”
#define AES_BLOCK_SIZE 128

RMuint32 * src,dst,iv,key;          // Pointers to source buffer, destination buffer, key and iv
RMuint32 size;                      // Size to process

// Fill in the src buffer, the iv, and the key.
...

// Decryption configuration
        struct portlib_cipher_ctx portctx;                  // Configuration structure

        portctx.mode = MODE_AES;                            // AES cipher
        portctx.pconfig.aes.aesmode = AES_CBC_DEC;          // CBC decryption mode

        portctx.size = (size/AES_BLOCK_SIZE) – 1;           // Number of AES blocks to process
        portctx.pconfig.aes.keysize = 16;                   // Key size in bytes
        portctx.pconfig.aes.blocksize = 16;                 // Block size in bytes
        memcpy(portctx.pconfig.aes.iv, iv, 4);              // Initialization vector
        memcpy(portctx.pconfig.aes.key, key, 4);            // Key

        portctx.src = src;                                  // Source address
        portctx.dst = dst;                                  // Destination address

// Trigger the cipher operation
        portlib_cipher_sync( &portctx);
```

VIII.2. Project 3

VIII.2.1. Code related the measurement of durations

```
// Part 1: Calculating the Average
avg = gbus_read_uint32(pgbus,REG_BASE_system_block+SYS_xtal_in_cnt);
for(i=0;i<1000;i++) status = xrpc_test_send_data( rua_aligned_addr);
avg= ( gbus_read_uint32(pgbus,REG_BASE_system_block+SYS_xtal_in_cnt)-avg);



// Part 2: Calculating Lowest & Highest

for(i=0;i<NUMBER_OF_CALLS;i++) {
        count= gbus_read_uint32(pgbus,REG_BASE_system_block+SYS_xtal_in_cnt);
        status = xrpc_test_send_data( rua_aligned_addr);
        count2=gbus_read_uint32(pgbus,REG_BASE_system_block+SYS_xtal_in_cnt);

        //printf("Call %d: %d us \n", i, (int)(count2-count)/27);
        results[i] = (count2-count) / 27;

        // Is it the lowest value so far ?
        if ( ((count2-count) < low) ) {
                low=count2-count;
        }

        // Is it the highest value so far ?
        if ( ((count2-count) > high) ) {
        // Handle counter reset
        if (((count>>31) == 1)&&((count2>>31) == 0)) status=0;
        else {
                high=count2-count;
        }
        }

}
```

VIII.2.2. Durations measured on Tango 2

With nothing else running on the board:
Highest 3970 us
Lowest 2154 us
Average 3419 us

While a simple program is running:
Highest 4007 us
Lowest 2157 us
Average 3462 us

While an AES encryption is going on:
Highest 77213 us
Lowest 2299 us
Average 4410 us

While a video is being played:
Highest 45425 us
Lowest 2292 us
Average 4113 us

VIII.2.3. Durations measured on Tango 3

With nothing else running on the board:
Highest 101 us
Lowest 58 us
Average 60 us

While a simple program is running:
Highest 217 us
Lowest 59 us
Average 61 us

While an AES encryption is going on:
Highest 4118 us
Lowest 58 us
Average 355 us

While a video is being played:
Highest 4356 us
Lowest 58 us
Average 63 us