Cryptographic Attestation of the Origin of Surveillance Images

Anton Eliasson
dat11ael@student.lu.se

Department of Electrical and Information Technology
Lund University

Axis Communications AB

Supervisors:
Martin Hell (LTH),
Johan Ekenstierna (Axis)
and Mikael Starvik (Axis)

Examiner: Thomas Johansson

2017

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

A method is devised that provides authentication and integrity protection for H.264 encoded surveillance video. A digital signature is created at the H.264 Network Abstraction Layer and included in the video stream, providing robustness against video container changes while remaining format compliant for compatibility with software that does not support the signing feature. The signature is created using asymmetric cryptography, which provides protection to both data in transit and at rest. The usage of asymmetric cryptography is compared to other methods of securing digital video and found to be the best approach for this application. Keys are unique per camera, allowing identification of the specific camera unit that created a particular video recording. A Public Key Infrastructure is described, where the camera vendor acts as a Certificate Authority.

A proof-of-concept implementation is developed for an Axis ARTPEC-6 development board. To establish that the platform is capable of operating the protocol in real time its cryptographic performance is first measured. The benchmark shows that for typical surveillance video the performance is sufficient. To protect the private part of the key used for signing even in the face of partial system intrusion, a memory access restriction feature that the platform provides is used. This feature is compared to the functions offered by standard Trusted Platform Modules. The concept itself is platform agnostic and can be implemented on any platform that handles H.264 video and offers similar security features.

Finally limitations of, as well as threats against, the concept are discussed and analysed. The protocol is considered a viable way of securing video and providing additional trustworthiness to the authenticity of surveillance video.

Foreword

I would like to thank Axis Communications for giving me the opportunity of working with this interesting master's dissertation. These people in particular have helped me during the course of this project: Fredrik Olsson, Johan Ekenstierna, Lars Persson, Linus Svensson, Mikael Starvik and Rabin Vincent. I also want to specifically thank my advisor at LTH, Martin Hell. To all of you, I'm grateful for your support during my work.

Contents

List of Figures

Chapter 1

Introduction

Surveillance images are often used for forensic purposes. Being possible evidence in court or in an insurance claim, it is crucial that the authenticity of images can be verified. The path between the actual camera and the forensic investigator who is to examine the surveillance images may be quite long and involve many people, causing it to be difficult to establish a chain of custody. The usage of digital signatures can improve this situation by using cryptography to authenticate digital images.

1.1 Digital signatures

In asymmetric cryptography, also known as public key cryptography, the key used is split into two parts. One is public and one is private. The public key is used for encrypting messages while the private key is used for decryption. Conversely, the private key is used to sign messages, while the public key is used to validate digital signatures.

Digital signatures are used for authentication in many different contexts. Because digital signatures can be validated at any time, by anyone that has the appropriate public key, they are useful for authenticating *data at rest*. They are for example used in authenticated electronic mail which may be stored for a long time before being read by the recipient. Digital signatures can also be useful for bootstrapping secure online communications protocols, where the sender and receiver negotiate a shared symmetric key. In the widely used Transport Layer Security (TLS) protocol digital signatures are used in the initial handshake in some configurations. Specifically, when *client authentication* is employed in TLS, which authenticates the client as well as the server, the client performs a signature in order to prove ownership of a private key. TLS is used in HTTPS to protect both the confidentiality of the data in transit using encryption, and also its integrity using a hash-based message authentication code (HMAC).

The requirements for surveillance image authentication are similar to that of TLS. The surveillance system should prove to its clients that the photos or videos are genuine and have not been tampered with. A trusted third party may be used to delegate trust in signing keys. The difference is that in this case data should also be protected while at rest in the client's custody, not only during transit. This implies that the digital signature must be stored along with the data it protects, either in the same container or as a separate data file, allowing it to be verified at any time.

1.2 Digital signatures in video surveillance

A digital signature can easily be created manually by the camera operator using common tools like GnuPG [1] before submitting the images to a forensic investigator. Some vendor-specific video management tools like Axis Companion also have this feature [2]. The sooner the signature is created though, the better, as this typically means that fewer people have handled the surveillance data before it is protected by a signature. Ideally, the signature should be created inside the surveillance camera itself, and this dissertation covers a method for that.

1.3 Goals

The major goal of this dissertation it to propose and evaluate a method for protecting the authenticity and integrity of surveillance video captured by Axis cameras in real time using a digital signature. The video data should be protected both in transit and at rest. Its integrity and authenticity should be easily verifiable by anyone who has access to the video itself. Through the use of public key cryptography the specific camera that created a recording can later be identified.

This project is purely technical. Although one goal of this project is to increase the trustworthiness of surveillance photography from Axis cameras in a forensic investigation, no legal claims will be made. The applicability of the contemplated digital signing features in court or as part of an insurance claim is outside the scope of this dissertation. It will also be designed as an optional feature. The lack of a signature in a video stream is more likely explained by not having added it in the first place, not by malicious removal. The verification software must be designed based on this assumption. Requirements on signature correctness can later be tightened once the feature has been deployed in all parts of a surveillance video system.

The dissertation describes:

- A protocol for transferring the signature in conjunction with the corresponding video.
- A proof-of-concept implementation of said method and protocol.
- How the secret key used to create the signature can be protected even in the face of partial intrusion.
- The short and well-defined list of parties that need to be trusted, and the threats considered.
- The cryptographic keys required and the corresponding infrastructure for handling the keys.

1.4 Threats against unprotected surveillance video

There are several conceivable attacks possible on surveillance video. Individual frames may be edited in an image manipulation program. There exist methods for detecting tampering of still images. For example, partial image manipulation can be detected by analysing compression artefacts [3]. Imperfections in the camera sensor cause a fixed patterned noise that is visible in the image and can be used to identify a camera given an image [4]. As these methods improve however, so does image manipulation software. This makes the idea of a cryptographic solution to this problem attractive, since it gives a clearer binary answer to the question of whether an image is genuine or not.

Image frames may also be deleted or reordered but otherwise unchanged. Imagine for example a set of images from a very still parking lot at night. It may be desirable to prove that nothing out of the ordinary occurred at this location. However, with little or no movement between image frames it may be difficult to visually discern a continuous video from a video where incriminating image frames have been removed from the sequence. Therefore it is important to not only authenticate frames individually, but also the order of image frames.

A sufficiently well funded adversary may stage an event with actors, producing a surveillance recording of an entirely fake incident in order to, for example, frame someone for a burglary. This is difficult to completely mitigate using only a technological solution. Including the time and date of the recording as well as the camera's identity in the produced video in a secure way may provide an indication of the origin of the video.

1.5 Related work

In 1996 Kelsey, Schneier and Hall [5] developed a protocol for an authenticated (hand-held) camera. It attempted to cryptographically tie a digital image not only to a time span but also to a physical location. It is not applicable as-is to surveillance images since it assumes that every image taken will be saved and used in hash chains. In surveillance most photography is usually thrown away by a computer algorithm or operator, and only the interesting parts are kept. Moreover, the protocol described mandates procedural changes since the camera must interact with a base station before and after each photographic event. However, the paper considers some interesting attacks that are also relevant to the method described in this dissertation.

High-end Canon hand-held digital cameras have an image signing feature, marketed as Original Decision Data. The security of this feature was broken by Elcomsoft researchers in 2010. The fundamental problem was that the secret key was stored insecurely and could be dumped as part of the firmware. Moreover, the same key was used in every camera of the same model [6].

MPEG-21 is an existing protocol that adds security features to multimedia. It is discussed in Section §3.2.

1.6 Structure of this report

In Chapter 2 the available hardware and software resources are described. Candidate cryptographic algorithms are introduced and the necessary parts of the H.264 video encoding standard are summarised.

Next, in Chapter 3 different approaches for authenticating digital video are discussed. Digital signatures are compared to digital watermarks and few remarks on message authentication codes are made.

The methodology for measuring the cryptographic performance of the target platform is described in Chapter 4 as well as the results thereof.

In Chapter 5 a protocol for authenticating digital video is proposed. A proof-of-concept prototype implementation of the protocol is described as well as insights acquired while developing it. The protocol is discussed and potential improvements are suggested.

The intended key management and a threat model is described in Chapter 6. This chapter defines the scope and limitations of the proposed protocol.

Finally Chapter 7 concludes the dissertation. It discusses potential future work and the concept's applicability to other video encoding implementations.

Chapter 2

Background

2.1 Targeted hardware

Development will be done on an ARTPEC platform of version 6.

ARTPEC-6 is an Axis developed custom platform based on an ARM main CPU (MCPU) which runs a Linux based operating system. While the main CPU supports ARM TrustZone, the operating system is currently not loaded via Secure Boot. Further, the customer (camera operator) typically has administrative access to the operating system. In fact, customer modifications to the default firmware with custom applications are encouraged and supported in Axis cameras. For these reasons the main operating system must be considered insecure in the context of signed images.

ARTPEC-6 also has several independent subsystems that together make up the *user defined logic* (UDL). The UDL contains all the video and image functionality and other Axis specific functions. Each subsystem consists of a helper CPU core known as an LCPU (*Local Central Processing Unit*) using the CRIS architecture as well as relevant hardware blocks for their task.

Typically surveillance video is streamed in real-time to either a professional video management system (VMS) or to a generic network attached storage (NAS) device. There is also the possibility of storing video locally in the camera on a removable memory card. This option is sometimes used in smaller installations that only have a minor number of surveillance cameras. The ARTPEC-6 platform supports both Motion-JPEG over HTTP and H.264 over RTP (Real-time Transport Protocol). Video compression is implemented in hardware, so modifications to the compression algorithms cannot be made using the current hardware platform. In the case where video is streamed to a VMS it is not typically transcoded or re-compressed, although it may be re-packed into a different container format [7].

2.1.1 User defined logic and the CRY subsystem

Most of the UDL subsystems are used in the pipelined processing of image data from the camera sensor. One subsystem, known as the *CRY*, is used for cryptographic operations. The CRY has exclusive access to the following hardware resources:

- A *Large Integer Arithmetic Unit* (LIAU). This is a hardware acceleration unit that can perform modular exponentiation on integers of up to 2049 bits in size. Modular exponentiation is an operation used in RSA calculations. The LIAU limits the RSA key length to 2048 bits (the 2049th bit is used for carry).

- A *True Random Number Generator* (TRNG). This unit is able to generate random data at an average bandwidth of 150 Mbps. It is used to seed the pseudorandom number generator in Linux, and also directly by the CRY LCPU in cryptographic operations that require random data as input.
- *A* 768 bit *One-Time Programmable* (OTP) memory which is not accessible from the main CPU. This can be used to store a per-chip unique symmetric key that protects the private part of an asymmetric key pair stored elsewhere. A program for this secure CPU can thus be created which allows it to sign data from the main CPU without revealing the private key. Because RSA keys that provide enough security are longer than 768 bits (see Section §2.3), they cannot be stored directly in the OTP memory.

The API between the MCPU and LCPUs is a set of command blocks with an *opcode* that specifies a defined operation and zero or more *parameters*. If the parameters are larger than a simple data type (for example, the data block to sign and the resulting signature), they are typically transferred by using a pointer parameter. The actual data is then copied to/from the LCPU via Direct Memory Access (DMA). Command blocks are processed by the LCPU in several stages of a software pipeline. The software pipeline has the following stages:

prep() initiates DMA transfers of additional data from main memory, if necessary for the command. The pipeline is advanced when the transfer is done.

exec() performs the actual work. If the work is expected to take long, a background thread is instead started here. If a background thread is used, it is responsible for advancing the pipeline when it is finished.

stat() initiates DMA transfers of the result back to main memory. The pipeline is advanced when the transfer is done.

done() means that the command is finished. A status code is sent back to the MCPU and the pipeline is then ready to accept a new command.

Command blocks may be queued in input and output FIFO queues, allowing the LCPUs and the MCPU to work independently without stalls caused by communication.

The LCPUs have no persistent program memory of their own, so their programs must be loaded during every camera boot. The CRY has 64 kB of RAM which is used for both code and run-time data. 16 kB are reserved for dynamic heap memory. A block diagram of the CRY and its communication channels with the MCPU is shown in Figure 2.1.

2.1.2 Main CPU subsystem

The image processing performed by the LCPUs terminate in a GStreamer element which communicate with the UDL through a Linux kernel driver. In the case of

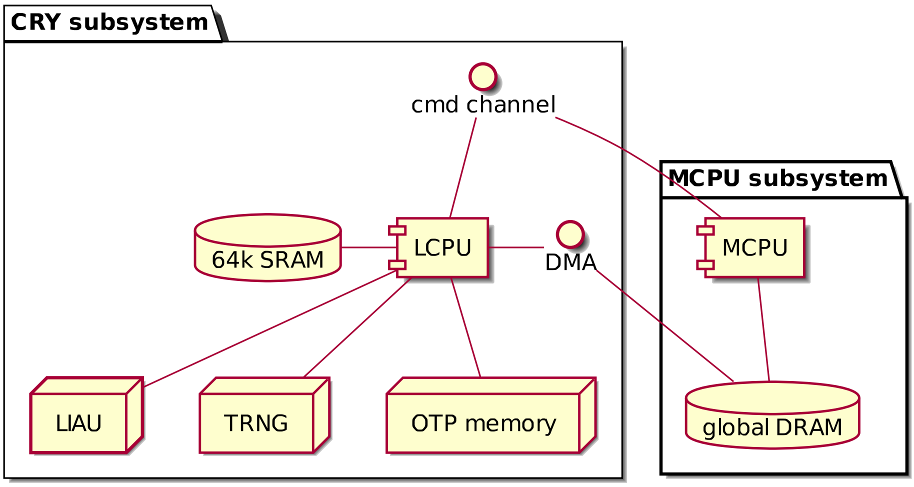

Figure 2.1: CRY block diagram.

H.264 the video is at this point compressed into individual image frames. Packing the compressed frames into a H.264 stream, multiplexing the resulting stream into a RTP container and sending it through the network is done in software in a GStreamer pipeline.

A simplified diagram of an example GStreamer pipeline is shown in Figure 2.2. GstArtpec6Src is the element that communicates with the UDL. In this case it is configured to output H.264 video at 1080p. Its source pad is connected to the sink pad of GstH264Sign. This is a new, blank element which currently does nothing except copy data from its input to its output. After that comes GstRtpH264Pay which encapsulates H.264 into an RTP stream. Finally the RTP stream is sent out on the network by GstUDPSink. In this example the video is streamed over UDP via the multicast IP address 224.2.0.1. UDP offers no guarantees regarding packet ordering or integrity. Errors during transmission will be seen by the receiver.

Normally RTP streams are set up by the RTSP protocol, which handles streaming sessions. RTSP provides configuration and teardown of RTP streams, user authentication and play and pause functionality. Apart from that, this is a reasonable example of how video is streamed from an ARTPEC-6 based camera.

2.2 Existing software resources

There exists a test program for the CRY that implements RSA cryptographic operations and expose them to the MCPU using LCPU commands. The program is based on the LibTomCrypt cryptographic library. According to its documentation [8],

> LibTomCrypt is a fairly comprehensive, modular and portable cryptographic toolkit that provides developers with a vast array of

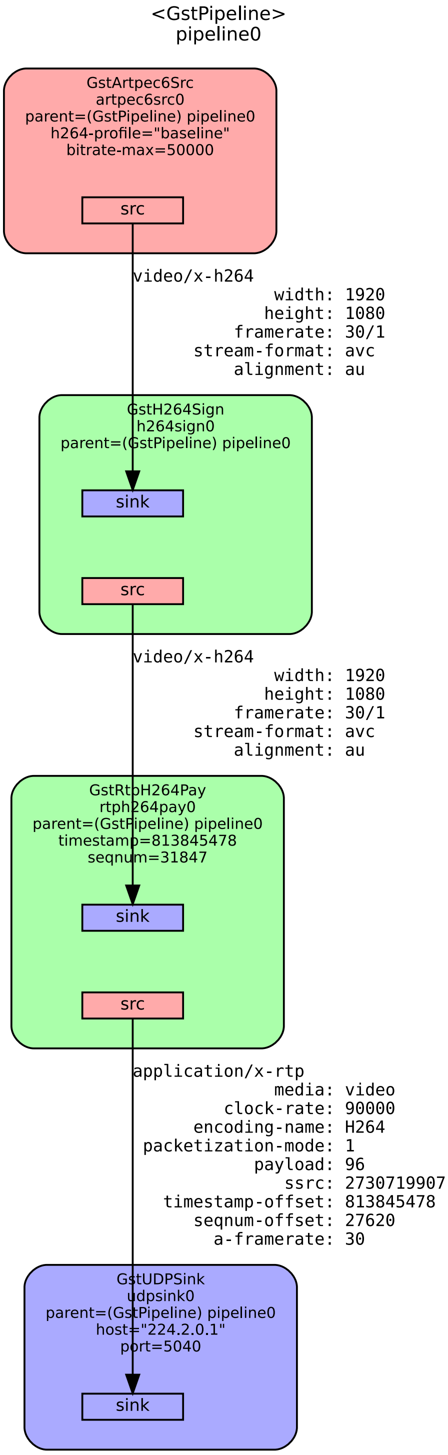

Figure 2.2: Example GStreamer pipeline.

> well known published block ciphers, one-way hash functions, chaining modes, pseudo-random number generators, public key cryptography and a plethora of other routines.

LTC has a very permissive open-source license (namely the WTFPL [9]), making it suitable for use in a proprietary program such as the CRY firmware. Currently no validation of the input data received from the MCPU is performed. Random data that is required during some operations (RSA OAEP and PSS padding) is taken from the hardware TRNG. The CRY program also implements an operation to supply random data to the MCPU.

2.3 Cryptographic algorithms

The RSA algorithm was initially chosen as the primary candidate for this project, due to existing hardware acceleration support in the platform. In this section is it discussed and compared to more modern elliptic curve cryptography algorithms that have recently increased in popularity.

2.3.1 RSA algorithm

One of the first algorithms that was used to implement digital signatures is RSA, first publicly described in 1978 [10]. It is now widely used in many applications. Though there are implementation issues that must be considered, the algorithm itself remains unbroken still, almost 40 years after its introduction. Its main problem is performance. In all practical applications the actual data is not signed directly. Instead a one-way hash function is used to digest the data into a short, non-reversible representation that is in turn signed. Still, the absolute cost of RSA is fairly high, meaning that it might be impractical to perform RSA in real time on performance limited hardware.

2.3.2 Elliptic curve cryptography

Elliptic curve cryptography is based on operations on elliptic curves over finite fields, in comparison to non-ECC algorithms (such as RSA) that operate directly on finite fields. The mathematics behind ECC will otherwise be summarised with the conclusion that smaller keys are required for the same level of security. The security of ECC is determined by the number of points on the curve, whereas the security of RSA is determined by the size of the modulus. ECC has been applied to digital signatures in two different schemes: ECDSA and EdDSA.

ECDSA is based on plain DSA and should, like DSA, only be used when a source of high quality random numbers is always present during signature generation. A lack of entropy in the random values consumed during signature generation may entirely compromise the private key. ECDSA is implemented in LibTomCrypt. It has been shown that although the mathematics remains sound, for some curves implementation aspects create security problems [11].

EdDSA was invented by Daniel J. Bernstein et. al. and is based on Schnorr signatures. EdDSA used on an elliptic curve nicknamed *Curve25519* (after the numbers in its prime $2^{255} - 19$) is known as *Ed25519*. The parameters were chosen to avoid elliptic curves covered by patents, while allowing efficient computations. Bernstein et. al. have published an Ed25519 reference implementation attributed to the public domain. Randomness is not consumed on signature generation, unlike ECDSA and regular DSA. Array indices and branch conditions do not rely on the private key, which provides resistance against certain side-channel attacks that rely for example on timing information [12].

2.3.2.1 Patents

The standard legal disclaimer applies to this section: The author of this dissertation has no formal education in patent law and can only cite what others have suggested.

The application of elliptic curves in cryptography was originally suggested independently by Koblitz and Miller in 1985 [13, 14]. While the idea is not new, it has not seen much use until recently much due to the fact that its legal situation has been unclear for a long time. Unlike RSA, the basic principle of elliptic curve cryptography was never patented in itself. However, many implementation techniques and practical optimisations of ECC *are* covered by patents [15]. Certicom corporation owns many patents and has had a strong claim, according to Bruce Schneider in 2007 [16].

Bernstein et. al. constructed their Ed25519 reference implementation specifically to avoid infringing on known patents. He dismisses several patents as being *trivial*, claims some patents to be invalid due to *prior art* and claims others to be irrelevant to useful applications of ECC. Some patents only apply to certain elliptic curves, which can be easily avoided. Finally, some patents on ECC have expired in the last few years [17, 18].

A well-known legal case on ECC is Certicom's 2007 lawsuit against Sony for their alleged infringement on Certicom patents in the implementation of Sony's DRM systems. The lawsuit was dismissed in 2009 due to both parties entering a settlement, and did not set a prejudice [19].

The sketchy conclusion of this short legal research is that there are strong indicators that elliptic curve cryptography signatures should be implementable without infringing on any valid, active software patent. The actual implementation in LibTomCrypt was not investigated at all (since it is not usable on the current hardware at any rate) and may use implementation techniques that are still patented. Ed25519 is recommended for further investigation and possible implementation.

2.3.3 Key lengths and relative security

The security level of a cryptographic algorithm is often expressed in terms of "equivalent bits of a symmetric key" [20].

As explained in section 2.1.1 the hardware acceleration unit limits the maximum length of RSA keys on the ARTPEC-6 platform to 2048 bits. 2048 bits is also the minimum recommended length by RSA Laboratories [21]. For these reasons a 2048 bit key length was chosen for the experiments in this project.

ECC not only requires shorter keys compared to RSA for equivalent security, they also scale linearly to the equivalent symmetric key. RSA keys scale exponentially, making RSA operations very computationally expensive when higher security is wanted. This makes digital signature schemes based on ECC attractive potential replacements for RSA.

Ed25519 public and private keys are 256 bits in size stored each and provide a security level of 128 bits. This is a bit stronger security than 2048 bit RSA keys provide. An RSA key with a 128 bit security level is approximately 3072 bits [22].

A 256 bit Ed25519 key could possibly fit into the fuse bits inside the CRY. A stored 2048 bit RSA key is larger than 2048 bits due to having several parameters where 2048 is only the size of the modulus. Such a key would not fit into the fuse bits. However, a unique per-chip symmetric key could be stored there which protects the private part of an RSA key stored in the unprotected main flash memory.

It should be noted that the key length is not the only factor in computational complexity. Careful selection of the key components can also have a significant impact on the practical efficiency in the implementation of a cryptosystem. RSA benefits from using a public exponent with a low hamming weight (number of non-zero bits), since modular exponentiation with such a number can be implemented very efficiently on binary computers. 65537 ($2^{16} + 1$) is commonly used. This allows for fast RSA signature verifications, even when large keys are used [23]. Similarly, in ECC it has been shown that the elliptic curve has a large impact on performance [24].

2.4 Overview of H.264

H.264 or formally MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC) is a video compression standard originally published in 2003 and improved in several iterations since. It was collaboratively developed by the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC JTC1 Moving Picture Experts Group (MPEG). Compared to previous MPEG compression standards, H.264 offers improved quality over the same bit rate.

The H.264 standard only specifies the *decoding* process. The *encoding* process is completely undefined, as long as the result fulfils all requirements the decoder sets. This approach leaves a lot of flexibility to the encoder implementer. It also means that there are often many ways to encode a given video sequence, and the decoder must be prepared to handle all the possible options. Inevitably, the standard will also specify optional features that no actual encoder will ever implement.

The H.264 standard defines a clear separation between its Video Coding Layer (VCL) and its Network Abstraction Layer (NAL). Stütz and Uhl [25] explain this separation:

> The VCL is responsible for creating a coded representation of the moving pictures, while the NAL formats these data and provides header information in a simple and effective fashion, i.e., a NAL unit header is not entropy coded. VCL data are organized into NAL units,

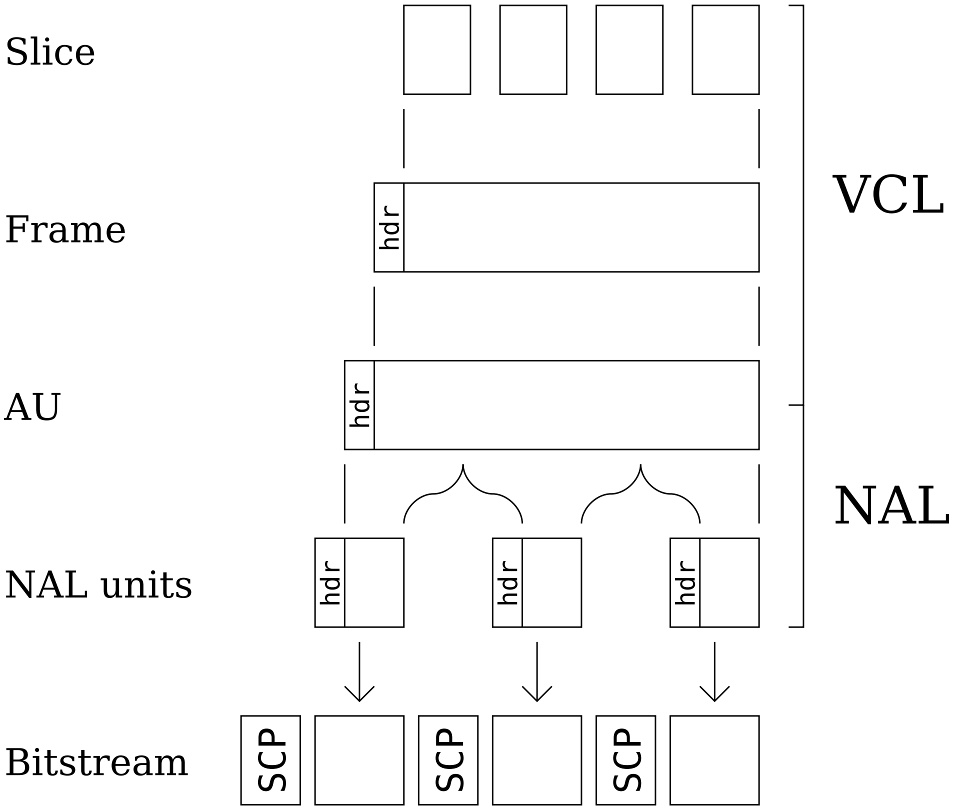

Figure 2.3: Layers in H.264. Slices are briefly described in section 2.4.5. SCP: *start code prefix*.

> which start with a one byte header. Most important is the type of a NAL unit (NUT) that is inferred from the NAL unit header [...]

This layered approach has allowed for fairly comprehensive extensions to the protocol, such as the Scalable Video Coding (SVC) enhancement. For this project only the NAL is relevant. The VCL will be explained only briefly in the next subsection. See Stütz and Uhl [25] and Wiegand et al. [26] for a more in-depth explanation of the VCL in H.264. The information in the rest of this section is based on those two papers.

An overview of the layers in H.264 is illustrated in Figure 2.3.

2.4.1 The video coding layer

Predictive video coding formats are based on the idea of periodic *spatially compressed intra-coded* video frames (frames that do not depend on any other frames) interleaved with *inter-coded* frames that may also be *temporally compressed* (frames that encode differences from a different frame). Video frames are distinguished by the following terms:

- I-frames are *intra-coded* as explained above.
- P-frames are *predictively coded* and contain motion-compensated differences relative to a previous frame.
- B-frames are *bi-predictively coded* and contain motion-compensated differences relative to *several* previous or *later* frames in the video stream.

The particular rules for B-frames and the particular differences between P- and B-frames are not relevant for this dissertation.

Frames are naturally grouped together into sequences starting with an I-frame and ending just before the next I-frame. Such sequences can be decoded individually. They are denominated *Coded Video Sequences* in the H.264 standard, which will be abbreviated as CVS in this dissertation. A Coded Video Sequence is equivalent to the term Group Of Pictures (GOP) used in other coding standards such as MPEG-2.

In H.264 frames are packed into *Access Units* (AU). An Access Unit contains a single coded frame as well as optional extra information relevant to the VCL. H.264 makes a difference between *Instantaneous Decoding Refresh* (IDR) Access Units, which are units containing an I-frame, and non-IDR AUs. The Access Unit constitutes the border between the VCL and the NAL.

2.4.2 The network abstraction layer

The Network Abstraction Layer is concerned with encoding Access Units in a way that can be represented as a stream of data bits. The basic unit of this layer is the NAL unit. Access Units from the VCL are possibly split, if they are too large, and prefixed with a header to be packed into NAL units. Other types of NAL units (non-VCL NAL units) contain video stream metadata and some units only exist to make the stream easier to parse by providing delimiters between other NAL units. Examples of metadata are *parameter sets* which describe how to decode VCL NAL units and Supplemental Enhancement Information (SEI) Units which are used for generic ("everything else") metadata. The contents of a NAL unit are identified by the NAL unit type field in the header.

Some NAL unit types are reserved for future use. Others are *unspecified* and may be used by implementations for any purpose. They will never be claimed by amendments to the H.264 specification and implementations conforming to the standard must ignore any *unspecified* NAL units that they do not recognise.

The specification defines an optional bitstream format (commonly known as the "Annex B format" after the appendix in the specification) which may be used to transport NAL units. In this format a *start code prefix* (0x000001) is inserted between each NAL unit. Any occurrence of 0x000001 inside the NAL unit content must be escaped by inserting an *emulation prevention byte* to prevent the decoder from treating it as a start code prefix.

Other transport methods are allowed, in which case the specification only considers the sequence of NAL units as the bitstream. A common method is to prefix each NAL unit with their size in bytes. This method is used in the MP4 container file format (formally MPEG-4 Part 14). In this scheme the number of bytes used for the size field itself cannot be inferred and must be specified elsewhere.

Some NAL units may be reordered in the bitstream. A prominent example is the B-frame which may refer to a future P-frame. To reduce buffering requirements during the decoding of these they are often *encoded* after the future P-frame they reference, even though they are to be *displayed* before the P-frame.

2.4.3 Parameter sets

Two types of *parameter sets* exist in H.264: Sequence Parameter Sets (SPS) and Picture Parameter Sets (PPS). Sequence Parameter Sets apply to entire Coded Video Sequences, while Picture Parameter Sets apply to individual pictures within Coded Video Sequences. Without the parameter sets, the VCL cannot be decoded. They may be included in the video stream packed into NAL units, or they may be conveyed out-of-band. The idea is that a more reliable channel may be used to transport the important parameter sets, while a less reliable (higher performance) channel may be used for the actual video frames. Alternatively, the parameter sets may be periodically repeated in the stream to provide redundancy. In theory, different Sequence Parameter Sets may be used for every Coded Video Sequence. In most practical applications, both SPS and PPS change very rarely.

In Axis cameras, parameter sets are basically static (changing only when the camera is reconfigured). They are generated by the *artpec6src* GStreamer element (see Figure 2.2) and transported in-band through the GStreamer pipeline. Over the network they are however typically transported out-of-band through the use of RTSP to set up a video streaming session.

2.4.4 Indexing NAL units

As it can take a substantial amount of time to generate an RSA signature, and stalling the image pipeline for that time would not be acceptable, the signature will have to be transported detached from the signed data. Assuming that signatures are computed piecewise in the H.264 Network Abstraction Layer, and that the signature units will be injected into the output video stream with a variable delay from the actual data, they must somehow contain a reference to this data. If there were a global sequence number that was monotonically incremented in every NAL unit, that would be an ideal candidate. A client that connects to an already (long) running live stream would then easily be able to tell the current position in the stream. Unfortunately the H.264 standard specifies no such thing. Because H.264 aims to be an efficient compression format, the standard is very much concerned with avoiding transmitting redundant information, or information that may be inferred by the decoder. The following are a few candidates found in the header of coded video slices that were considered but ultimately rejected as possible NAL unit sequence numbers for this project:

- A *frame_num* parameter which increases in *decoding* order. These only identify frames within Coded Video Sequences however, and are reset to 0 on the next IDR.

- An optional Picture Order Count parameter which increases in *display* order. When this is present, it is split into two parts: LSB and MSB (Least and Most Significant Bits, respectively). The LSB identify slices within Coded Video Sequences, while the MSB identify Coded Video Sequences. However, only the LSB are encoded into the slice headers. MSB are inferred by the decoder and are incremented when the LSB reach their maximum value and roll over to 0.
- There is a concept of *reference picture marking*, of both *long* and *short term*. In both cases a picture is tagged with an index, which tells the decoder that it should hold on to the picture even after displaying it. Short term reference pictures will be automatically deleted when the decoder's buffer of reference pictures is full. Long term reference pictures must be manually deleted by an explicit command in the bitstream. This approach was rejected due to a combination of being too intrusive and too complicated to implement. Every image frame would have to be modified to be tagged with a reference and in the case of *long term marking* additional removal commands would have to be injected into the bitstream. The implications of monopolising this feature were deemed too uncertain.

An alternative to absolute indexing is to use relative numbering. Typically, the signature would be encoded very closely, but not directly adjacent, to the data it refers to. This would result in a low delta counted in any of the primitive units in a H.264 bitstream, which can be efficiently Exp-Golomb encoded. Exponential Golomb coding (Exp-Golomb for short) is a method of encoding numbers using a variable number of bits. The length of each coded word can be inferred by the receiver. It is similar to Huffman coding, except that it does not use a translation table. Exp-Golomb coding is widely used in the H.264 standard to encode unbounded numbers.

2.4.5 Concluding remarks

This section is a simplified explanation of the concepts in H.264. It ignores the *slice* concept which essentially divides frames further into *slice groups* where each *slice* may be encoded differently. For example, a frame may contain an intra-coded *I* slice as well as a predictive *P* slice. It follows that an I-frame is a frame consisting only of I-slices.

Chapter 3

Authenticating surveillance video

To authenticate pictures and video several techniques are possible. Cryptography can be used through digital signatures to create a mathematical proof that either a piece of data is authentic, or a very difficult ("infeasible") computational operation has been performed. Drawing similarities to the analogue world of photographies printed on paper, another technique comes to mind: Watermarks. Watermarked paper is traditionally used to indicate that an important document, such as a passport or a banknote, is authentic and an original copy. While it provides confidence in the origin of a document, another important property is that a watermark cannot be easily removed from a piece of paper. Physical watermarks have a digital counterpart, and it is this latter property that is most commonly featured in a digital watermark. The applications of cryptography and digital watermarks overlap somewhat, but they solve very different problems as will be explained.

3.1 Digital watermarks

Digital watermarks are concerned with the embedding of additional information about a work, into the same work. The original work without the watermark is referred to as the *cover work,* because it hides or "covers" a secret message. Typically the watermark should be *imperceptible* under normal conditions, but easily readable by someone who is authorised to. If it is perceptible to the consumer it will have altered the work, possibly devaluing it.

In contrast, cryptography typically assumes a reliable communications channel and provides confidentiality, integrity, authentication and/or non-repudiation [27].

By embedding extra data into the original work itself through a watermarking algorithm, the size does not increase. This is advantageous in certain situations, where the size of a message is constrained and cannot change. However, as the watermark constitutes extra data that must be encoded in the fixed space, some other data must be removed. This is simply a consequence of Shannon's coding theorem [1]. There are several methods for choosing which data to replace with the

[1] Informally stated by MacKay [28, ch. 4] as:

> N i.i.d. random variables each with entropy $H(X)$ can be compressed into more than $N\,H(X)$ bits with negligible risk of information loss, as $N \to \infty$;

watermarking bits. One method, arguably the simplest, is to encode it into the highest frequency parts of the cover work, as they are often the least perceptible to the consumer. Because the information that was there before is overwritten, this leads to a quality degradation in the technical sense, whether it is perceptible or not.

Digital watermarking algorithms usually (but not always) share the trait with its physical counterpart that they are difficult to remove or render unusable once embedded, without damaging the original content. This trait is called the *robustness* of a watermarking algorithm. A robust watermark remains decodable through transformations and signal processing operations, both malign and benign, of the watermarked content. Examples of such transformations are temporal and spatial filtering, lossy compression and geometric distortions (such as rotation, translation and scaling) [29]. Thus, robust watermarking is a *transport protocol* for additional information to the cover work, as argued by Cox et al. [30]. It allows for reliable (to some degree) transmission of data over an unreliable medium.

Many watermarking algorithms employ a keying function to spread the encoded data over a large region (in time, space, and/or frequency), similar to spread-spectrum encoding. Only using the same key can the watermark be decoded. Without the key the watermark may be detectable, but not decodable or removable without adding so much distortion that the work will be perceptibly damaged. However, with the ability to decode the watermark using the key also comes the ability to remove it. Thus, the key must be kept secret. The capability of read-but-not-remove is a highly sought after, but difficult to achieve, property in many digital watermarking schemes [27].

A digital watermark may also be designed to be *fragile*. A fragile watermark is undecodable even after a subtle change to the watermarked work. Such watermarks can be used for authentication. A watermark that is designed to be robust against benign (such as lossy compression) but not malign transformations is referred to as *semi-fragile* [31, ch. 10]. Identifying what constitutes a valid transformation and what isn't and designing a suitable semi-fragile watermarking algorithm is not a trivial task [32].

A watermark can be applied before, during or after compression. This is referred to as the *domain*. A fragile watermark must be applied either during or after compression, as it would otherwise be immediately destroyed. A robust or semi-fragile watermark on the other hand can be applied before compression if deemed suitable.

3.1.1 Uses of digital watermarks

In *Digital Watermarking and Steganography*, Cox et al. [29] list a few common use-cases for digital watermarking. These are exemplified briefly below; for further information refer to the book.

- Signalling start and stop codes to a machine processing information content

but conversely, if they are compressed into fewer than $N\,H(X)$ bits it is virtually certain that information will be lost.

intended for human consumption, such as broadcast TV.

- Identifying and proving ownership of a work.
- *Transaction tracking*, also referred to as *traitor tracing* or *fingerprinting*. This is used for example to identify the person leaking a confidential work, such as a preview of an unreleased film.
- *Device* and *copy control*, where the watermark instructs a playback device how a work may be consumed for DRM purposes. Device manufacturers are then forced to implement decoding and enforcing of the device controlling watermarks by law or policy.
- *Content authentication*, where the content receiver desires confidence in the integrity and origin of a work.

These use-cases place different demands on the watermarking algorithm.

In recent years digital watermarking techniques specifically applied to H.264 compressed video have been widely researched. Many of these are robust algorithms that aim to provide content fingerprinting for DRM of streamed video [33].

3.1.2 Digital watermarks for content authentication

As described in previous sections, digital watermarking can imaginably be used for a wide variety of purposes. In literature, many authors blur the boundary between digital watermarks and cryptography. For instance, a cryptographic hash that is stored inside otherwise unused data fields in individual video frames could arguably be called a fragile watermark even though the fragility is a property of the hash function, not the watermarking function [34]. Kim and Afif [35] cite several examples of watermarking schemes that were later proven broken because they were not based on cryptography theory at all, or did not utilise cryptographic primitives properly.

For this reason, robust watermarking is seen only as a transport medium in this dissertation, insufficient for authenticating video in itself. This argument is also made by Cox, Doërr and Furon [27].

3.2 Digital signatures

For a general introduction to digital signatures, refer to Chapter 1.

Hellwagner et al. [36] discuss different levels in a network stack at which encryption of streaming multimedia content may be implemented. The same arguments apply to content authentication and are reiterated below.

- Security at the *transport level* protects *data in transit*. Examples are SRTP (Secure Real-time Transport Protocol), TLS and IPsec. This approach provides no security to *data at rest* however, making it unsuitable in this project.
- Introduce an intermediary meta-format between the application (H.264 in this case) and transport layer. MPEG-21 is such a standard for video. It was published originally in 2001 [37], revised in 2004 [38] and has seen several

additions since then. It is a wide-spanning framework for multimedia that can provide content confidentiality, integrity and authentication as well as DRM features. MPEG-21 has been used to implement authentication for H.264 video specifically, for example by Iqbal, Shirmohammadi and Zhao [39]. A problem with introducing another layer is that it is a non-backwards compatible change in that all parts of the system must support the new layer. As this would be a major architectural change for limited benefit compared to the other approaches, authentication using MPEG-21 was not pursued in this project.

- Security at the *codec format level* may provide protection to both *data in transit* and *data at rest*. Many multimedia codecs support user-defined data fields in their bitstreams. By signing standard data fields and appending the signature in a custom field the stream remains compatible with compliant decoders. This is a non-standard approach however, unless the codec specifies support for embedding digital signatures. Interpretation of the user-defined fields containing the signature will be vendor-specific. Another disadvantage with this approach is that it is specific to a codec. Other data that may be part of the same multimedia stream, such as audio, is not protected.

Since codec format level security offers protection to stored data, it was pursued in this project. In the case of H.264 Axis owns the entire chain between the camera and the video management system and can afford a non-standard addition to H.264 for an optional feature such as this.

3.2.1 Digital signatures for content authentication

For a general introduction to the H.264/AVC stream format, see Section §2.4.

Stütz and Uhl [25] describes an approach for H.264 encryption, which is later improved by Hellwagner et al. [36]. In this approach every NAL unit in the H.264 stream is individually encrypted. The NAL unit type is then rewritten to one of the nine *unspecified* types. The resulting stream is still H.264 format compliant because according to the H.264/AVC specification, decoders must ignore NAL units with unspecified unit types in the bitstream. In the original paper, AES in ECB mode is used while in the follow-up paper the security of the technique is improved by using AES in CTR mode instead.

Almost exactly the same technique can be used to authenticate the bitstream by hashing and signing NAL units instead of encrypting them and storing the signature in an *unspecified* type. The data stream size will increase somewhat by the addition of the signatures, but that is not an issue in this application.

To authenticate the order of frames the hash can be calculated over both the current and the previous image frame. If a frame is missing from the stream the following hash will then be incorrectly calculated, causing the signature verification to fail. To avoid hashing the same data twice, this approach can be optimised to hash the hashes of the current and previous frame instead of the full data.

This will only allow the verifier to determine whether *any* image frame in the video stream has been corrupted or lost. It will not be able to determine the number of invalid frames. For that a more elaborate authentication scheme is required, such as a tree of hashes that is authenticated on multiple levels. This is outside the scope of this dissertation.

3.3 Message authentication codes

Message authentication codes are commonly used to protect data in transit. For streaming video they are for example used in SRTP. MACs can be used to protect data at rest, but because they are computed and verified using the same (symmetric) key, they can only be verified by the same party who issued them [40]. In order for a MAC to provide long-term end-to-end data integrity protection to surveillance video, MAC keys would have to be stored securely during normal operation and only be used when suspicion is already awoken. This limits their use for the purposes of this project, and were not pursued further.

Chapter 4

Cryptographic performance

To determine the possibilities of using the CRY for signing images in real time, an early step was to benchmark its cryptographic performance. The easiest way to do this is to simply use a fix length cryptographic key to sign a fix length piece of data and measure the time it takes. To obtain a precise measurement without overhead from communication inside the ARTPEC platform, the time should ideally be measured inside the LCPU creating the signature itself.

4.1 Measuring LCPU resource usage

The time required for an LCPU to complete a specific job can be assessed by measuring the time spent in the *exec()* stage of the software pipeline (as described in section 2.1.1) while the job is executed. A difficulty that arose was that the pipeline stages are performed in an interrupt context, with most other interrupts disabled. This is normally not a problem, since a well-behaved computation intensive command of an LCPU application works by only setting up a background thread in the pipeline *exec()* stage and using a callback function to advance the pipeline when the computation has finished. However, this ruled out using a periodic interrupt to count discrete time steps (as is done in the *lcpu_timer* part of LCPU OS). Fortunately, every LCPU has a 64-bit counter register that is automatically incremented by hardware at 100 MHz (10 nanosecond steps) which can be read out in software. This counter was used to measure the time spent inside the *exec()* stage of a particular software pipeline, which can later be read out by the MCPU. A program could then be written that simply blocks the *exec()* stage of a pipeline for the duration of a signature creation, which is in fact how the existing proof-of-concept CRY program worked. Implementation was also complicated somewhat due to the fact that debug printing does not work in interrupt contexts. The debug print routine enters data into a ring buffer, which is cleared out and sent to the MCPU by an interrupt. This causes debug messages to stack up and only be sent to the MCPU after the LCPU exits interrupt context, i.e. when *exec()* is done.

4.2 Comparing RSA and ECDSA

RSA signing performance could be readily measured. Using the aforementioned method, generation of a 2048-bit RSA signature over 32 bytes of data (the size of a SHA256 hash) was measured at 292 ms with hardware accelerated modular exponentiation. With hardware acceleration disabled, a signature takes 619 ms to generate.

An unexpected obstacle appeared during the implementation of ECDSA support however – the LCPU ran out of heap memory! To measure the memory required to create an ECDSA signature using LibTomCrypt, a few simple test programs were created and run on the x86_64-based development PC. As only the general order of magnitude in signature generation time is of consequence in this dissertation, the test programs could also provide an indication of approximate CPU usage in instruction count. The test programs are listed in Appendix A. For the tests, LTC 1.17 and TFM 0.13 was used. The compiler used was GCC 5.4.0. Since the development PC did not have a fast true RNG, the pseudo-RNG *Yarrow* was used instead. The Valgrind tool *Massif* [41] was used to measure CPU and heap memory usage over the duration of the program runs. To filter out memory allocations caused by I/O and key import/export functions (which would not be done in the actual implementation), the functions `_IO_file_doallocate` and `rsa_import` were ignored using Massif's `--ignore-fn` option.

4.2.1 Results and discussion

The results from Massif were graphed using the Massif-Visualizer [42] application.

The exact number of instructions is obviously not comparable to the number of instructions it would take on an ARTPEC LCPU, since Intel x86 and CRIS are vastly different CPU architectures. The instruction count is also a very crude approximation of the actual time required to run a program, as instructions are associated with very different costs. Assuming that the bulk of the instructions used by LTC to implement RSA and ECDSA are similar, only different in quantity, the ratio between the instructions required for the same operation using RSA and ECC can be expected to be similar across CPU architectures.

The graphs show that ECDSA signature generation is about three times faster than (non-hardware accelerated) RSA signature generation. This is likely not enough to make a practical difference in the implementation of digital signatures for real-time video. The graphs also show that ECDSA requires almost 30 kB of dynamic memory, making it infeasible to run on the current hardware.

These measurements were performed before researching the difference between ECDSA and EdDSA. EdDSA was not benchmarked as it is not currently implemented on the CRY and may perform differently.

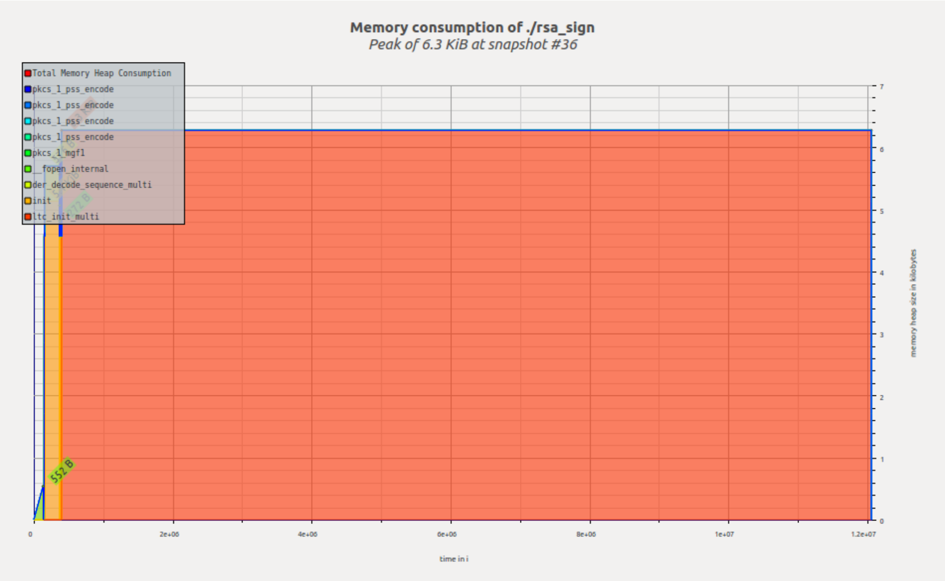

Figure 4.1: Heap memory usage of rsa_sign over time. The X axis shows the time in instructions executed and the Y axis shows the memory consumption in kB.

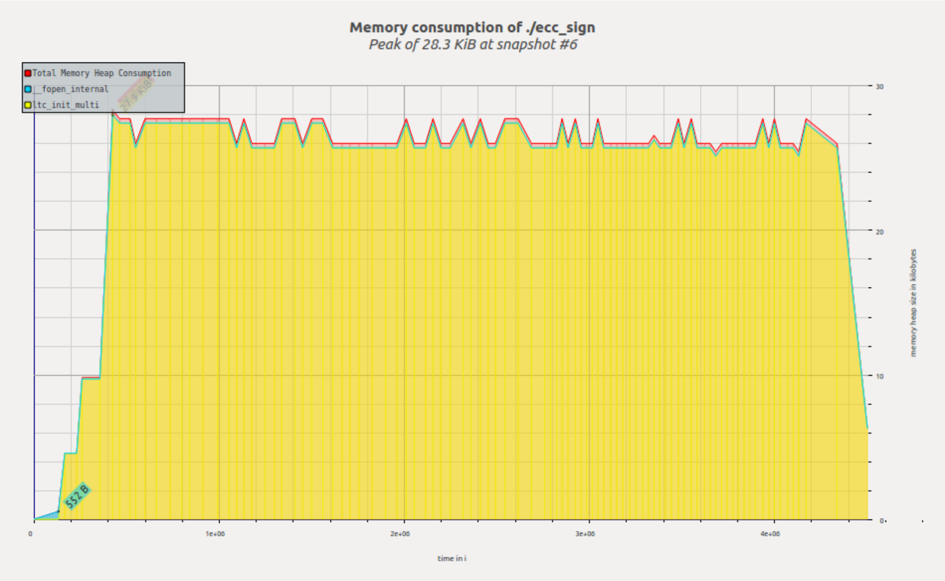

Figure 4.2: Heap memory usage of ecc_sign over time. The X axis shows the time in instructions executed and the Y axis shows the memory consumption in kB.

Chapter 5

Implementation

A simple protocol was devised that describes how signatures of H.264 compressed video are created, transferred and verified. Due to time constraints only parts of said protocol were implemented. Additional improvements that were found out during prototyping or left out in the initial protocol design to keep the project scope down are also listed. Since the contemplated improvements have not been tested in an actual proof-of-concept implementation it is likely that they will have practical issues that are yet to be found.

5.1 Assumptions

In this initial protocol, the following assumptions were made:

- There is only one video stream with the signing feature active at any given time. The implementation may assume exclusive access to the signing hardware.
- H.264 parameter sets are conveyed embedded in the video stream.

5.2 Protocol

For simplicity, all integers are encoded as 8-bit numbers in this first version of the protocol. An obvious improvement would be to use Exp-Golomb coding instead.

5.2.1 Signing

The signing consists of the following tasks:

- For every Coded Video Sequence k:
    - Create a rolling SHA-256 hash over all NAL units in the Coded Video Sequence to produce $Hcvs_k$.
    - When the Coded Video Sequence ends (i.e. when the next CVS starts):
        - Finalise the rolling hash $Hcvs_k$.
        - Concatenate $Hcvs_k$ with $Hcvs_{k-1}$ and hash it with SHA-256 to produce $Hsign_k$.

  - Asynchronously order CRY to sign $Hsign_k$ using the RSA algorithm and PSS message padding, producing S_k. If the CRY is busy calculating an older signature then place the new job in a queue.

- For every signature S_k computed by CRY:
  - Create a new NAL unit with a custom (unspecified) NAL unit type dubbed *signing NAL unit* with the following contents:
    - An identifier that specifies that this *signing NAL unit* contains a signature coded in the format described in this section.
    - A reference to the Coded Video Sequence k that was signed, in the form of a *delay* measured in Coded Video Sequences.
    - The actual signature S_k.
  - Insert the new NAL unit into the stream between the current Coded Video Sequence and the next.

This protocol does not modify data in the video stream; it only injects additional data packets at the Network Abstraction Layer. Since signatures are computed in order and all signatures take the same amount of time to compute [1], it follows that they are also inserted into the output stream in order.

5.2.2 Key transfer

In order to verify the signature, the receiver needs the corresponding public key. To also verify that the key can be trusted, a certificate signed by a trusted third party is used. The key and certificate could be conveyed out-of-band like the parameter sets (see section 2.4.3), but to make the video recording self-contained they can easily be embedded into the stream itself in the same way. This will also allow a live viewer to verify the integrity of the streamed video with no special set-up other than a streaming client with support for signature verification.

Only the case where the key and certificate are embedded into the stream is considered in this protocol. The NAL unit type *signing NAL unit* that was devised in the previous section is re-used, with a different identifier that specifies that the content is a key or a certificate. No timestamp or *delay* parameter is needed. The NAL unit thus contain only the following:

- Key/certificate identifier.
- Key/certificate data.

The key format is a DER encoded PKCS #1 RSAPublicKey ASN.1 structure [43], with *emulation prevention bytes* inserted to avoid illegal byte sequences in the resulting byte stream (as described in section 2.4.2). The certificate format remains unspecified in this first protocol iteration. The key/certificate NAL unit is injected into the stream before each SPS encountered. This approach adapts to the same redundancy that the application chooses for the parameter sets.

[1] Any cryptographic implementation for which this is not true is leaking information either about the key or the message data.

The parameters used for signing (hash function, padding scheme and salt length) are not included in the key/certificate NAL unit, since they are hard coded to the above values in this protocol version.

5.2.3 Verification

The verification program first looks for the *signing NAL unit* containing the packed key/certificate. In the case of a certificate it is verified against the verification program's configured list of Certificate Authorities. Then, a rolling hash is created over all NAL units belonging to a Coded Video Sequence. When the CVS ends (i.e. when the next CVS starts; same rule as described above), finalise the hash and add it to a list of hashes to verify. When a *signing NAL unit* containing a signature is encountered, use it to verify the stored hash in the list as indexed by the *delay* parameter. The signature should always refer to the oldest hash in the list; anything else is an error condition:

- If the signature refers to an older hash (higher *delay* value) than is available in the list, then the signature should be discarded and the user interface indicate a verification failure. This can happen legitimately if the client just joined a live video stream and did not see the data the delayed signature refers to. It could also happen if an entire IDR NAL unit is lost, so that the verifier does not detect that a new Coded Video Sequence has started. The verifier will eventually "catch up", but one or more signatures will fail to verify (up to the number in the *delay* parameter).
- If the signature refer to a hash with a lower *delay* value than is available in the list it means that the video stream is missing signatures. This could theoretically happen if the camera device is overloaded and is refraining from signing all its data. It could also happen if video authentication is implemented as an optional feature with an on/off switch, and the feature is turned on while a client is already streaming (unauthenticated) video. To catch up, the verifier should drop hashes from its verification list until the *delay* parameter of the current signature refers to the oldest hash while indicating verification failure.

For practical purposes the list of hashes can be bounded in size to the maximum expected value of the *delay* parameter. Signature failures are treated as warnings displayed to the video consumer, along with the seek time of the failure. In the case of recorded video the seek time is relative to the beginning of the file, while in the case of live viewing the seek time can be relative to when the streaming session started.

5.3 Proof-of-concept implementation

The signing part of the protocol was implemented as a GStreamer element. This approach allowed the program to be included in the existing software on the ARTPEC-6 platform with basically no changes. It also allowed fast prototyping by offloading irrelevant code like I/O and NAL unit parsing to standard GStreamer plugins and libraries. The proof-of-concept implementation can concern itself almost exclusively with implementing the signing protocol devised in this chapter, apart from some glue code. The message digest functionality in OpenSSL is used to compute SHA256 hashes in the signing GStreamer element. The implementation communicates directly with the CRY via a debug interface exposed in *sysfs* by a very simple Linux kernel driver. In this early proof-of-concept the private signing key is manually uploaded to the CRY, while the public key is supplied to the signing element to be packaged and injected into the H.264 stream. Other than the key to use, the CRY requires (and accepts) no particular run-time configuration, as all parameters are hard-coded in its firmware.

The verification program was also implemented as a GStreamer element, for much the same reasons. By making it a pass-through element (*transform* in GStreamer terminology), its output can be connected to a H.264 decoder and display in order to demonstrate signature validation in real-time during playback. OpenSSL's implementation of both SHA256 message digests and RSA signature validation is used in the verification element. Signature failures are reported through GStreamer's logging framework as warning messages.

OpenSSL was chosen because it is widely used and was assumed to be a mostly correct implementation of the cryptographic functions used in this project.[2] It was also already included in the operating system running on ARTPEC-6.

All parts of the protocol above are implemented in the proof-of-concept prototype, with the following exceptions:

- No certificate validation is done. The verification program simply accepts whatever key is included in the video stream.
- Signature validation failures are reported, but not timestamped in the logging message.
- There are bugs in the implementation that cause the video stream to be corrupted when signing live video. For whatever reason it only works on recorded video, making it somewhat useless in its current state.

Figure 2.2 shows a typical usage of the signing element in a GStreamer pipeline.

The GStreamer element will continue to copy all input data to its output unmodified, only inserting additional packets. A sequence diagram of the signing process is shown in Figure 5.1. The time it takes to sign a hash according to the wall clock is fixed, but measured in NAL units passed through it may differ.

5.4 Discussion and possible improvements

5.4.1 Implementation issues

It was found that using a custom NAL unit type is not as robust as initially assumed. Inserting the key/certificate NAL unit before the SPS was not a successful approach. The GStreamer element h264parse was found to discard all data preceding the first SPS, causing the signing key to get lost. This is not unreasonable,

[2] Although it is *still* not entirely audited [44].

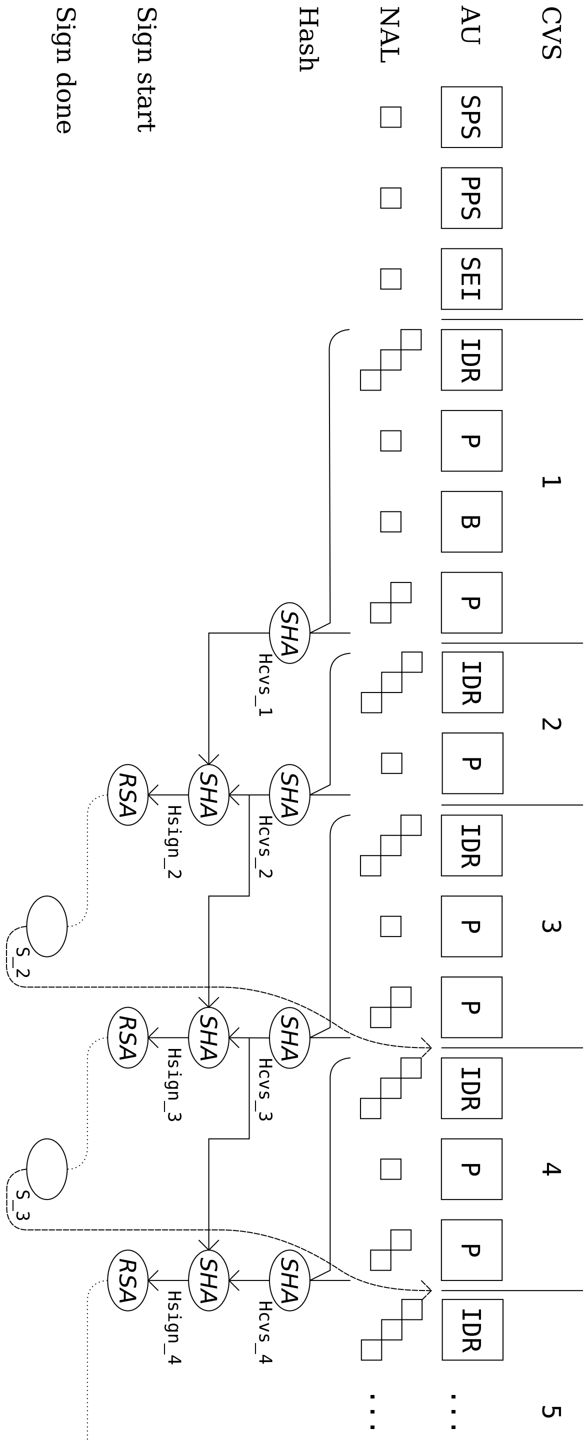

Figure 5.1: Sequence diagram of the signing process.

as any video data encountered before the parameter sets are available cannot possibly be decoded. As a workaround, the certificate unit could be inserted after the initial parameter sets instead.

During testing a basic GStreamer pipeline (shown in Figure 2.2) was used to stream video over RTP. It was found that the GStreamer packing element *rtph264pay* performs certain parsing of incoming data and discards unspecified NAL units. Again, this is a legal response according to the standard (section 2.4.2), but it was not expected. To work around this issue the *rtph264pay* element could be modified. Alternatively, instead of using an unspecified NAL unit, the custom signature data could be packed into the standard SEI NAL unit which also allows for user-defined data.

5.4.2 Signing parameters

In this initial protocol version all signing parameters are hard coded. To allow for variation in the parameters, as well as support for other signing algorithms than RSA, the parameters should be included in the key/certificate structure embedded in the stream. Standard protocols for encoding this information exist, such as the Cryptographic Message Syntax (CMS) [45]. This protocol is used for example in the S/MIME standard for secure e-mail [46].

5.4.2.1 Timestamping

To further increase the credibility of recorded surveillance video, a secure timestamp could be included. The timestamp would have to be authenticated along with the video data. This is not a difficult improvement to realise; simply include a coded representation of the current time in the rolling hash. Alternatively, the Cryptographic Message Syntax supports embedding the signing time in a standardised way.

If the camera operators themselves are considered adversaries, the timestamp must come from a secure real-time clock, one which can be access controlled. The current hardware only offers a standard real time clock chip which is not designed to be tamper-resistant against the camera operator and thus gives limited assurance in this use-case.

5.4.3 Fault tolerance

This approach couples the division of Coded Video Sequences with the signature granularity, making it somewhat easier to reason about. It follows that the signature validation process will be delayed by one Coded Video Sequence, and the very first Coded Video Sequence will not be verifiable. In a lossy network individual NAL units may be lost, causing distortion to the video and the signature verification to fail. The video will be fully restored when the next IDR frame is received, i.e. when the next Coded Video Sequence starts. At this point signature verification can also re-start, although it will take an additional Coded Video Sequence before the first data can be verified again.

Because the public key is embedded in the stream at the same rate as the H.264 parameter sets, new clients joining a live stream will also be able to start verifying the integrity of streamed video as soon as they are able to decode it. Again, this makes it easier to reason about when the signature should be expected to verify successfully (when video shows on the screen, the signature should be valid). Of course, more elaborate hash trees are possible, which may allow further verification of the Coded Video Sequence order even in the face of packet loss.

5.4.4 Platform dependence

The prototype implemented is directly tied to the ARTPEC-6 and its CRY. It currently uses debug interfaces to directly communicate with the CRY through a very simple Linux kernel driver and is quite fragile in the face of changes to the operating system. This approach would not work in a production environment.

The concept itself, however, is not dependent on the platform at all. The CRY is used in a fashion similar to a Trusted Platform Module (TPM). It could be replaced by an actual TPM or an implementation performing a similar function.

Another possible approach is to abstract the underlying secure hardware using the Linux kernel Crypto API. The Crypto API is primarily used in parts of the kernel that uses cryptography, for example encrypted file systems and network protocols. Cryptographic operations may be provided by software or hardware implementations [47]. In 2010 a user-space interface for the Crypto API based on Netlink sockets was proposed and subsequently merged. It is commonly known as AF_ALG after the new socket address family it adds to Netlink. This allows user-space applications to access cryptographic hardware through a standardised interface. Currently it only supports one-way hashes and symmetric key operations (and combinations of these, such as HMACs and AEAD) [48]. A patch that adds support for asymmetric key operations such as digital signatures to the user-space API has been proposed to the Linux Kernel Mailing List but is not yet merged as of January 2017 [49]. Stephan Mueller, one of the authors of the Crypto API, has written a user-space library *libkcapi* that can be used to communicate with the kernel using the new Netlink API [50].

Axis could wait for this patch to be merged, possibly helping the effort along by fixing the problems found during review on the Linux Kernel Mailing List. A Crypto API driver for the ARTPEC-6 CRY could be implemented, allowing it to be used in a mostly platform-independent manner.

Chapter 6

Use-cases and threat model

6.1 Key management

The devised image authentication protocol assumes the following properties of key management:

Each camera is assigned an RSA key pair during manufacturing. The private key is encrypted with a symmetric key which is stored in memory which is inaccessible to the main operating system running on the camera. In the ARTPEC-6 platform this can be the OTP memory inside CRY. The public key and the encrypted private key are stored at a location in main flash which is persistent across factory resets and firmware updates.

Camera specific keys have a certificate that is signed by the camera vendor during manufacturing, to attest that the key was in fact issued by the vendor. The certificate could potentially have limited period of validity, requiring certificate renewal. However, since the camera vendor has little possibility of verifying the camera integrity once it is out of their hands, this is not expected to offer more security than issuing certificates with unlimited validity. The camera vendor acts as a Certificate Authority and its root certificate should be bundled with the part of its Video Management System that is able to validate authenticated video. It is of utmost importance that the private CA key is kept secret.

6.2 Threat model

With the hardware limitations detailed in Chapter 2, the number of trusted parties in the contemplated system can be as few as two: the camera vendor and the camera operator. The vendor is responsible for implementing the camera software securely, while the operator is responsible for installing the camera as intended and protecting the physical and remote access to the camera.

The rest of this section lists some of the most important threats and practical problems to the concept, and how they are either dismissed or mitigated.

6.2.1 Transmission errors and video editing

Any errors during transmission of the streaming video and its embedded signature will cause the verification to fail. Any video post-processing done after the signature is applied (such as cropping or masking) will also invalidate the signature. The VMS or storage system must store the video as it is received. Even then, signature verification may fail occasionally due to transmission errors, since the video is usually streamed over UDP. The frequency of transmission errors varies depending on the network reliability, which may range from very high (in the case of a local unsaturated network) to lower (due to network saturation or when streaming over the internet).

Even when video is streamed over TCP which uses an error detecting checksum, some errors will still slip through. Statistically, at least one error in 65536 will not be detected (since the checksum has 16 bits). It has been shown that for some systematic (non-random) errors this may occur much more often [51].

The signature is easily stripped from video recordings, maliciously or because some part of the video management system does not support it. In this case they should be assigned the same trustworthiness as recordings from a camera without the authentication feature.

6.2.2 Camera software bugs

The software running on the camera can, and is even likely to have bugs. These bugs may allow an adversary to assume partial or full control of the camera. An adversary with full administrative access will at the very least be able to forge signatures by directly communicating with the CRY from the main operating system. By storing the secret key in the CRY's OTP memory it is nominally protected against malicious code running on the main CPU. However, since the firmware running on the CRY is not authenticated on the ARTPEC-6 platform, an adversary could construct an LCPU program that reads out the secret key from OTP memory and load it onto the CRY.

This is not a trivial attack. The software development kit for the LCPUs and precise documentation of their architecture are not publicly accessible. This does not make the attack infeasible to a determined adversary though, only more involved.

6.2.3 Malicious operator

A malicious operator has the same privileges as described in the previous threat. Administrator login credentials should be carefully guarded to avoid giving unauthorised users access to the camera.

6.2.4 Physical tampering

An adversary may replace the image sensor with a fake sensor that provides the camera with counterfeit image data, or simply place a screen in front of the camera. This is a viable attack against any surveillance camera, analogue or digital, and is essentially impossible to mitigate using software. Whether it is practical or not, it serves to define the scope of the video authentication feature: The authentication starts inside the camera and can only attest what the camera sees, not what actually happened in the scene.

6.2.5 Recording of a staged event

Again, the camera has no way of analysing the validity of the incident it is capturing on video. The signature does however serve to identify the specific camera through the public key fingerprint. The public key fingerprint may in turn be used to identify the person or organisation who owns the camera, by having the camera vendor maintain a list of which camera it sold to which customer. Alternatively, the fingerprint may simply be used to indicate a deviation from the camera which is normally monitoring a certain location. Including a timestamp in the signature serves to authenticate the time of recording.

6.2.6 Replay attack

Unless signature verification failures are treated as hard-failures, a man-in-the-middle attacker could easily mount a replay attack. A repeated signed video sequence will be piecewise correct. The signature verification would only fail occasionally as the video is looped. Including a timestamp in the signature would completely mitigate this attack. Another approach is to use a counter and have the receiver verify that the counter value is never repeated or decremented.

6.2.7 Broken cryptography

There are several parts to this:

- The signing and hashing algorithms used may some day be proven broken. This does not happen to cryptographic algorithms every day, but it has happened before. The widely used MD5 hash function was found to have design flaws as early as 1996 [52] and a practical attack against SSL certificates using MD5 hashes was demonstrated in 2008 [53].

- The cryptographic primitives may be used in an insecure way in the invented protocol. This is not unlikely, and so the protocol should be properly peer-reviewed before putting it to use. As an example, version 2.0 of the Secure Sockets Layer protocol (later renamed to Transport Layer Security) has been found to have severe design flaws. An IETF RFC recommends against its use since 2011 [54].

- The implementations of the cryptographic primitives may be broken, either OpenSSL or LibTomCrypt. An example of such a problem affected the version of OpenSSL shipped with the Debian Linux distribution between 2006 and 2008. A packaging bug caused the entropy used while generating keys to be limited to 15 bits [55]. Another prominent example is the now well-known OpenSSL “Heartbleed” bug discovered in 2014, where memory that potentially contains fragments of the private key could be leaked to a remote attacker [56]. It should be noted that LibTomCrypt is not nearly as widely used as OpenSSL and as such has likely had less public scrutiny.

6.3 Discussion: On not trusting the operator

With additional changes to the camera hardware and usage policy, even the camera operator can be taken out of the list of trusted parties, so that only the camera vendor remains. Several significant changes need to be made:

- All software running on the camera as well as firmware updates must be authenticated to have been issued by the vendor using a *secure/verified boot* scheme.
- The keys used to authenticate the software must be tamper resistant and unchangeable by the operator.
- The operator can no longer have unlimited administrative access to the camera.
- If a timestamp is included in the video signature, it must be derived from a secure non-user-settable clock.

Not only does this pose significant technical hurdles, it also raises the question of whether it is an ethical thing to do. Essentially, the camera operator no longer owns the camera. One could argue that this redistribution of administrative power also moves the liability of keeping the camera system secure and functional entirely to the camera vendor. This may be reasonable for a business model where products are leased rather than sold, although that is only speculation by the author.

Chapter 7

Conclusions

The issues described in the previous chapter place an upper bound on the additional trustworthiness that the invented protocol brings to a video recording. The video authentication protocol as implemented on the ARTPEC-6 platform can give a strong indication, but not a guarantee, that a signed surveillance recording was created by an Axis camera and that it has not been manipulated afterwards. Because of the many normal cases that may invalidate the signature, such failures cannot be treated as fatal. This is unlike for example S/MIME secure e-mail, TLS and other protocols that are only used over reliable communication channels, where a failed message integrity check is exceptional and should be treated fatally. Rather, a discontinuity in the chain of valid signatures should be treated as a warning in the video management system, which points out a timestamp in the video where the viewer should pay close attention to investigate the reason for the verification failure.

By changing some of the initial assumptions, the requirements on signature correctness be tightened. At a minimum, the video should be streamed over a reliable channel and the signing feature should be deployed to most parts of a surveillance system. To estimate the extent of remaining legitimate signature verification failures, interoperability testing between actual products should then be performed.

The protocol reduces the list of trusted parties to the camera operator and vendor. It emphasises the need to keep the camera software secure and up to date to avoid signature forgery and theft of cryptographic keys.

7.1 Future work

7.1.1 Signature transport encoding

Further investigation is needed regarding the viability of using an unspecified NAL unit to encode the signature. With this approach modifications need to be done to several other parts that handle the video stream (such as the GstRtpH264Pay element). Alternatively the signature could possibly be encoded in an SEI NAL unit as described in section 5.4.1.

7.1.2 Combining with encryption

The digital signature scheme devised could easily be combined with the encryption method described by Hellwagner et al. [36] to provide both confidentiality and integrity protection of the video stream. A proven method for combining signing and encryption should be used, as otherwise the result may have security deficiencies [57].

7.1.3 Axis ZipStream

ZipStream is an implementation of H.264 invented by Axis. It aims to reduce bit rate required for surveillance video by dynamically varying the quality of the video depending on its contents. ZipStream varies compression ratio, frame rate and length of Coded Video Sequences by estimating how interesting the video is, while always producing H.264 that is format compliant [58]. Since the protocol for video authentication described in this dissertation operates on the Network Abstraction Layer independently of the contents of the video, it is also applicable for ZipStream encoded video. ZipStream may however affect the protocol's fault tolerance and time to recovery in the face of transmission errors, as it is related to the CVS length.

7.1.4 H.265

H.265, also known as High Efficiency Video Codec (HEVC) or formally MPEG-H Part 2, is an evolution of H.264 which offers additional features and higher data compression at the same level of video quality. It shares the concepts of Video Coding Layer, Network Abstraction Layer and Access Units with H.264 and so the protocol for video signing should be easily adaptable for H.265. At the general level specified in this dissertation, the protocol is exactly the same. The implementation will be slightly different, because the NAL unit header has a different format and the NAL unit types have different identifiers.

7.1.5 Hardware changes

Although the CRY has more capabilities, in the implementation done as part of this dissertation it is used in the same way as a Trusted Platform Module. The protocol described could easily be implemented on a different platform. A secure key storage could for example be served by a standard TPM.

Bibliography

[1] GnuPG Project. *The GNU Privacy Guard*. 2016-08-18. URL: https://gnupg.org/ (visited on 2016-09-22).

[2] Axis Communications. *Axis Companion Video Management Software Data Sheet*. URL: http://www.axis.com/files/sales/ds_axis_companion_59749_en_1604.pdf (visited on 2016-09-21).

[3] T. Pevny and J. Fridrich. "Detection of Double-Compression in JPEG Images for Applications in Steganography". In: *IEEE Transactions on Information Forensics and Security* 3.2 (2008-06), pp. 247–258. ISSN: 1556-6013. DOI: 10.1109/TIFS.2008.922456.

[4] Jan Lukáš, Jessica Fridrich and Miroslav Goljan. "Detecting Digital Image Forgeries Using Sensor Pattern Noise". In: Proc. SPIE. Vol. 6072. 2006, 60720Y–60720Y–11. DOI: 10.1117/12.640109.

[5] J. Kelsey, B. Schneier and C. Hall. "An Authenticated Camera". In: *Computer Security Applications Conference, 1996., 12th Annual.* Computer Security Applications Conference, 1996., 12th Annual. 1996-12, pp. 24–30. DOI: 10.1109/CSAC.1996.569666.

[6] Dmitry Sklyarov. "Forging Canon Original Decision Data". CONFidence 2.0. Prague, Czech Republic, 2010-11-29/2010-11-30. URL: http://www.elcomsoft.com/presentations/Forging_Canon_Original_Decision_Data.pdf (visited on 2016-09-05).

[7] *E-Mail Correspondence*. In collab. with Fredrik Brozén. 2016-10-18.

[8] *Libtom/Libtomcrypt*. URL: https://github.com/libtom/libtomcrypt (visited on 2016-12-19).

[9] *WTFPL — Do What the Fuck You Want to Public License*. URL: http://www.wtfpl.net/ (visited on 2016-12-19).

[10] R. L. Rivest, A. Shamir and L. Adleman. “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. In: *Commun. ACM* 21.2 (1978-02), pp. 120–126. ISSN: 0001-0782. DOI: `10.1145/359340.359342`.

[11] Daniel J. Bernstein and Tanja Lange. *SafeCurves: Introduction.* 2014-01-19. URL: `http://safecurves.cr.yp.to/` (visited on 2016-12-20).

[12] Daniel J. Bernstein. *Introduction.* 2011-09-27. URL: `http://ed25519.cr.yp.to/` (visited on 2016-12-20).

[13] Neal Koblitz. “Elliptic Curve Cryptosystems”. In: *Mathematics of Computation* 48.177 (1987), pp. 203–209. ISSN: 0025-5718, 1088-6842. DOI: `10.1090/S0025-5718-1987-0866109-5`. URL: `http://www.ams.org/mcom/1987-48-177/S0025-5718-1987-0866109-5/` (visited on 2016-12-20).

[14] Victor S. Miller. “Use of Elliptic Curves in Cryptography”. In: *Advances in Cryptology — CRYPTO ’85 Proceedings.* Conference on the Theory and Application of Cryptographic Techniques. Springer, Berlin, Heidelberg, 1985-08-18, pp. 417–426. DOI: `10.1007/3-540-39799-X_31`. URL: `http://link.springer.com/chapter/10.1007/3-540-39799-X_31` (visited on 2016-12-20).

[15] RSA Laboratories. *FAQ 6.3.4 Are Elliptic Curve Cryptosystems Patented? (Archived Copy).* URL: `https://web.archive.org/web/20070919013834/http://www.rsa.com/rsalabs/node.asp?id=2325` (visited on 2007-09-19).

[16] Betanews. *Certicom Patent Suit Against Sony Threatens to Unravel AACS.* 2007-05-30T20:32:45+00:00. URL: `http://betanews.com/2007/05/30/certicom-patent-suit-against-sony-threatens-to-unravel-aacs/` (visited on 2016-12-20).

[17] Daniel J. Bernstein. *Irrelevant Patents on Elliptic-Curve Cryptography.* URL: `http://cr.yp.to/ecdh/patents.html` (visited on 2016-12-20).

[18] Daniel J. Bernstein. *Software.* 2015-06-11. URL: `http://ed25519.cr.yp.to/software.html` (visited on 2016-12-20).

[19] Justia Dockets & Filings.
Certicom Corporation et Al v. Sony Corporation et. Al., Filing 112. 2009-05-27.
URL: https://docs.justia.com/cases/federal/district-courts/texas/txedce/2:2007cv00216/103383/112 (visited on 2016-12-20).

[20] Arjen K. Lenstra.
Key Lengths: Contribution to The Handbook of Information Security. 2004.

[21] Robert D. Silverman. *Has the RSA Algorithm Been Compromised as a Result of Bernstein's Paper?* 2002-04-08. URL: https://www.emc.com/emc-plus/rsa-labs/historical/has-the-rsa-algorithm-been-compromised.htm (visited on 2017-01-05).

[22] Elaine Barker.
Recommendation for Key Management Part 1: General.
NIST SP 800-57pt1r4.
National Institute of Standards and Technology, 2016-01.
URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf (visited on 2017-01-09).

[23] Dan Boneh et al.
"Twenty Years of Attacks on the RSA Cryptosystem".
In: *Notices of the AMS* 46.2 (1999-02), pp. 203–213.
URL: http://www.ams.org/notices/199902/boneh.pdf (visited on 2017-02-01).

[24] Adam Langley. *ImperialViolet - What a Difference a Prime Makes.* 2010-12-21. URL: https://www.imperialviolet.org/2010/12/21/eccspeed.html (visited on 2017-02-01).

[25] T. Stütz and A. Uhl.
"Format-Compliant Encryption of H.264/AVC and SVC". In: *Tenth IEEE International Symposium on Multimedia, 2008. ISM 2008.* Tenth IEEE International Symposium on Multimedia, 2008. ISM 2008. 2008-12, pp. 446–451. DOI: 10.1109/ISM.2008.52.

[26] T. Wiegand et al.
"Overview of the H.264/AVC Video Coding Standard".
In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (2003-07), pp. 560–576. ISSN: 1051-8215.
DOI: 10.1109/TCSVT.2003.815165.

[27] Ingemar J. Cox, Gwenaël Doërr and Teddy Furon. "Watermarking Is Not Cryptography". In: *Digital Watermarking.* Ed. by Yun Qing Shi and Byeungwoo Jeon. Lecture Notes in Computer Science 4283. Springer Berlin Heidelberg, 2006-11-08, pp. 1–15. ISBN: 978-3-540-48825-5 978-3-540-48827-9. DOI: 10.1007/11922841_1. URL: https://link.springer.com/chapter/10.1007%2F11922841_1 (visited on 2016-10-24).

[28] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms.* New York, NY, USA: Cambridge University Press, 2002. ISBN: 978-0-521-64298-9.

[29] Ingemar J. Cox et al. "Chapter 2 - Applications and Properties". In: *Digital Watermarking and Steganography (Second Edition).* The Morgan Kaufmann Series in Multimedia Information and Systems. Burlington: Morgan Kaufmann, 2008, pp. 15–59. ISBN: 978-0-12-372585-1. URL: http://www.sciencedirect.com/science/article/pii/B978012372585150005X (visited on 2016-10-24).

[30] Ingemar J. Cox et al. "Chapter 10 - Watermark Security". In: *Digital Watermarking and Steganography (Second Edition).* The Morgan Kaufmann Series in Multimedia Information and Systems. Burlington: Morgan Kaufmann, 2008, pp. 335–374. ISBN: 978-0-12-372585-1. URL: http://www.sciencedirect.com/science/article/pii/B9780123725851500139 (visited on 2016-10-24).

[31] Borko Furht, Edin Muharemagic and Daniel Socek. *Multimedia Encryption and Watermarking.* Vol. 28. Multimedia Systems and Applications. Springer US, 2005. ISBN: 978-0-387-24425-9. URL: http://link.springer.com/10.1007/b136785 (visited on 2016-10-24).

[32] Ingemar J. Cox et al. "Chapter 11 - Content Authentication". In: *Digital Watermarking and Steganography (Second Edition).* The Morgan Kaufmann Series in Multimedia Information and Systems. Burlington: Morgan Kaufmann, 2008, pp. 375–423. ISBN: 978-0-12-372585-1. URL: http://www.sciencedirect.com/science/article/pii/B9780123725851500140 (visited on 2016-10-24).

[33] T. Stütz, F. Autrusseau and A. Uhl. "Non-Blind Structure-Preserving Substitution Watermarking of H.264/CAVLC Inter-Frames". In: *IEEE Transactions on Multimedia* 16.5 (2014-08), pp. 1337–1349. ISSN: 1520-9210. DOI: 10.1109/TMM.2014.2310595.

[34] Dima Pröfrock et al. "H.264/AVC Video Authentication Using Skipped Macroblocks for an Erasable Watermark". In: 2005-07-01, p. 59604C. DOI: 10.1117/12.632709. URL: http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.632709 (visited on 2016-11-01).

[35] Hae Yong Kim and Amir Afif. "A Secure Authentication Watermarking for Halftone and Binary Images". In: *International Journal of Imaging Systems and Technology* 14.4 (2004), pp. 147–152. ISSN: 0899-9457, 1098-1098. DOI: 10.1002/ima.20018. URL: http://doi.wiley.com/10.1002/ima.20018 (visited on 2016-11-01).

[36] Hermann Hellwagner et al. "Efficient In-Network Adaptation of Encrypted H.264/SVC Content". In: *Image Commun.* 24.9 (2009-10), pp. 740–758. ISSN: 0923-5965. DOI: 10.1016/j.image.2009.07.002. URL: http://dx.doi.org/10.1016/j.image.2009.07.002 (visited on 2016-10-13).

[37] ISO. *ISO/IEC TR 21000-1:2001 - Information Technology – Multimedia Framework (MPEG-21) – Part 1: Vision, Technologies and Strategy.* 2001-12-15. URL: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=30819 (visited on 2016-10-26).

[38] ISO. *ISO/IEC TR 21000-1:2004 - Information Technology – Multimedia Framework (MPEG-21) – Part 1: Vision, Technologies and Strategy.* 2004-11-01. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=40611 (visited on 2016-10-26).

[39] R. Iqbal, S. Shirmohammadi and J. Zhao. "Hard Authentication of H.264 Video Applying MPEG-21 Generic Bitstream Syntax Description (gBSD)". In: *2007 IEEE International Conference on Multimedia and Expo.*

2007 IEEE International Conference on Multimedia and Expo. 2007-07, pp. 875–878. DOI: 10.1109/ICME.2007.4284790.

[40] RSA Laboratories. *FAQ 2.1.7 What Are Message Authentication Codes? (Archived Copy).* URL: http://web.archive.org/web/20040617121326/http://www.rsasecurity.com/rsalabs/node.asp?id=2177 (visited on 2004-06-17).

[41] Valgrind Developers. *Valgrind Massif Manual.* URL: http://valgrind.org/docs/manual/ms-manual.html (visited on 2016-10-10).

[42] linux-apps.com. *Massif Visualizer.* URL: https://www.linux-apps.com/p/1127160/ (visited on 2016-10-10).

[43] Kathleen M. Moriarty et al. *RFC 8017 – PKCS #1: RSA Cryptography Specifications Version 2.2.* 2016-11. URL: https://tools.ietf.org/html/rfc8017 (visited on 2017-01-18).

[44] *Is OpenSSL Audited yet? CII/OCAP/NCC Audit in Progress...* URL: http://isopensslauditedyet.com/ (visited on 2017-01-19).

[45] Russ Housley. *RFC 5652 – Cryptographic Message Syntax (CMS).* 2009-09. URL: https://tools.ietf.org/html/rfc5652#section-5 (visited on 2017-01-13).

[46] Blake Ramsdell and Sean Turner. *RFC 5751 – Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification.* 2010-01. URL: https://tools.ietf.org/html/rfc5751 (visited on 2017-01-13).

[47] Stephan Mueller and Marek Vasut. *Linux Kernel Crypto API — The Linux Kernel Documentation.* URL: https://www.kernel.org/doc/html/latest/crypto/index.html (visited on 2017-01-11).

[48] Jake Edge. *A Netlink-Based User-Space Crypto API [LWN.Net].* 2010-10-20. URL: https://lwn.net/Articles/410763/ (visited on 2017-01-11).

[49] Tadeusz Struk. *[PATCH v8 0/6] Crypto: Algif - Add Akcipher (and Replies).* E-Mail. Thu Jun 23 2016 - 18:55:31 EST. URL: http://lkml.iu.edu/hypermail/linux/kernel/1606.2/06903.html (visited on 2017-01-11).

[50] Stephan Mueller. *Libkcapi - Linux Kernel Crypto API User Space Interface Library.* 2016-12-11. URL: `http://www.chronox.de/libkcapi.html` (visited on 2017-01-11).

[51] J. Stone et al. "Performance of Checksums and CRCs over Real Data". In: *IEEE/ACM Transactions on Networking* 6.5 (1998-10), pp. 529–543. ISSN: 1063-6692. DOI: `10.1109/90.731187`.

[52] Hans Dobbertin. "The Status of MD5 After a Recent Attack". In: *RSA Laboratories CryptoBytes* (Summer 1996). URL: `ftp://ftp.arnes.si/packages/crypto-tools/rsa.com/cryptobytes/crypto2n2.pdf.gz` (visited on 2017-02-01).

[53] Alexander Sotirov et al. "MD5 Considered Harmful Today". In: 25th Chaos Communication Congress. 2008-12-30.

[54] Sean Turner and Tim Polk. *RFC 6176 - Prohibiting Secure Sockets Layer (SSL) Version 2.0.* 2011-03. URL: `https://tools.ietf.org/html/rfc6176` (visited on 2017-02-01).

[55] *CVE-2008-0166.* URL: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166` (visited on 2017-02-01).

[56] OpenSSL. *OpenSSL Security Advisory: TLS Heartbeat Read Overrun (CVE-2014-0160).* 2014-04-07. URL: `https://www.openssl.org/news/secadv/20140407.txt` (visited on 2017-02-01).

[57] Donald T. Davis. "Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML". In: Usenix Tech. Conf. 2001. 2001. URL: `http://world.std.com/~dtd/sign_encrypt/sign_encrypt7.ps` (visited on 2017-01-24).

[58] Axis Communications. *Axis' Zipstream Technology | Axis Communications.* URL: `https://www.axis.com/se/sv/technologies/zipstream` (visited on 2017-01-24).

[59] Glyph & Cog, LLC. *Pdfdetach Man Page.* 2011-08-15. URL: `http://www.dsm.fordham.edu/cgi-bin/man-cgi.pl?topic=pdfdetach` (visited on 2016-10-07).

Appendix A

LibTomCrypt test programs

In Section §4.2 a set of test programs was devised to enable performance testing of LibTomCrypt on a PC. Code listings for these programs follow. As text copied from PDFs often lose much of its structure depending on the viewer, the code in these listings are also attached to this document as source code files. Extract them using a PDF viewer that supports attachments, or using a specialised tool like *pdfdetach* [59].

To run the test programs, first install TomsFastMath somehow. The version bundled with any modern Linux distribution is probably fine. There are special requirements on LibTomCrypt however; it must be built with TFM support (TFM_DESC defined). Build it from source like this:

```
CFLAGS=-DTFM_DESC make
```

Then place the test program source files in a subdirectory inside the LTC source directory. Build the programs individually like this:

```
gcc -g -o rsa_gen rsa_gen.c test.c ../libtomcrypt.a -DTFM_DESC -I
    ../src/headers -ltfm
```

Exchange rsa_gen with the names of the other test programs to build them as well. All programs read their input data from *standard input* and writes to *standard output*.

Listing A.1: test.h: header for test.c

```
#include <stdio.h>
#include <tomcrypt.h>

prng_state yarrow_prng;

int hash_idx, prng_idx;

void init(void);
```

Listing A.2: test.c: initialisation code common for all test programs

```
#include "test.h"

```

```
void init(void)
{
    int err;

    ltc_mp = tfm_desc;

    register_hash(&sha1_desc);
    register_prng(&yarrow_desc);

    err = rng_make_prng(128, find_prng("yarrow"), &yarrow_prng, NULL
        );
    if (err != CRYPT_OK) {
        fprintf(stderr, "rng_make_prng failed: %s\n",
                error_to_string(err));
        exit(1);
    }

    hash_idx = find_hash("sha256");
    prng_idx = find_prng("yarrow");
    if (hash_idx == -1 || prng_idx == -1) {
        fprintf(stderr, "rsa_test requires sha256 and yarrow\n");
        exit(1);
    }
}
```

Listing A.3: rsa_gen.c: generates a 2048-bit RSA key and writes it to stdout DER encoded

```
#include "test.h"

int main(void)
{
    unsigned char out[2048];
    rsa_key key;
    unsigned long len;
    size_t wr;
    int err;

    init();

    err = rsa_make_key(&yarrow_prng, prng_idx, 2048 / 8, 65537, &key
        );
    if (err != CRYPT_OK) {
        fprintf(stderr, "make_key err %d: %s\n", err,
            error_to_string(err));
        exit(1);
    }
    len = sizeof(out);
    err = rsa_export(out, &len, PK_PRIVATE, &key);
    if (err != CRYPT_OK) {
        fprintf(stderr, "export err %d: %s\n", err, error_to_string(
            err));
        exit(1);
    }

    wr = fwrite(out, 1, len, stdout);
    if (wr != len) {
        fprintf(stderr, "failed to write key to stdout, %zu/%lu
            bytes written\n", wr, len);
        exit(1);
    }

    rsa_free(&key);
    return 0;
}
```

Listing A.4: rsa_sign.c: inputs a DER encoded private key and creates a signature of a SHA256 hashed 32-byte null-string

```
#include "test.h"

int main(void)
{
    unsigned char in[2048], out[256], data[32] = {0}, hash[32];
    rsa_key key;
    unsigned long len;
    size_t wr, rd;
    int err;
    hash_state md;

    init();

    len = sizeof(in);
    rd = fread(in, 1, len, stdin);
    if (rd == 0) {
        fprintf(stderr, "failed to read key from stdin\n");
        exit(1);
    }

    err = rsa_import(in, rd, &key);
    if (err != CRYPT_OK) {
        fprintf(stderr, "import err %d: %s\n", err, error_to_string(
            err));
        exit(1);
    }
    sha256_init(&md);
    sha256_process(&md, data, sizeof(data));
    sha256_done(&md, hash);

    len = sizeof(out);
    err = rsa_sign_hash(hash, sizeof(hash), out, &len, &yarrow_prng,
         prng_idx, hash_idx, 0, &key);
    if (err != CRYPT_OK) {
        fprintf(stderr, "sign_hash err %d: %s\n", err,
            error_to_string(err));
        exit(1);
    }

    wr = fwrite(out, 1, len, stdout);
    if (wr != len) {
        fprintf(stderr, "failed to write signature to stdout, %zu/%
            lu bytes written\n", wr, len);
        exit(1);
    }

    rsa_free(&key);
}
```

Listing A.5: ecc_gen.c: generates a key on the 256-bit NIST curve and writes it to stdout in LTC special format

```
#include "test.h"

int main(void)
{
    unsigned char out[256];
    ecc_key key;
    unsigned long len;
    size_t wr;
    int err;

    init();

    err = ecc_make_key(&yarrow_prng, prng_idx, 32, &key);
    if (err != CRYPT_OK) {
        fprintf(stderr, "make_key err %d: %s\n", err,
            error_to_string(err));
        exit(1);
    }

    len = sizeof(out);
    err = ecc_export(out, &len, PK_PRIVATE, &key);
    if (err != CRYPT_OK) {
        fprintf(stderr, "export err %d: %s\n", err, error_to_string(
            err));
        exit(1);
    }

    wr = fwrite(out, 1, len, stdout);
    if (wr != len) {
        fprintf(stderr, "failed to write key to stdout, %zu/%lu
            bytes written\n", wr, len);
        exit(1);
    }

    ecc_free(&key);
    return 0;
}
```

Listing A.6: ecc_sign.c: inputs a LTC special format private key and creates a signature of a SHA256 hashed 32-byte null string

```
#include "test.h"

int main(void)
{
    unsigned char in[256], out[128], data[32] = {0};
    ecc_key key;
    unsigned long len;
    size_t wr, rd;
    int err;

    init();

    len = sizeof(in);
    rd = fread(in, 1, len, stdin);
    if (rd == 0) {
        fprintf(stderr, "failed to read key from stdin\n");
        exit(1);
    }

    err = ecc_import(in, rd, &key);
    if (err != CRYPT_OK) {
        fprintf(stderr, "import err %d: %s\n", err, error_to_string(
            err));
        exit(1);
    }

    len = sizeof(out);
    err = ecc_sign_hash(data, sizeof(data), out, &len, &yarrow_prng,
         prng_idx, &key);
    if (err != CRYPT_OK) {
        fprintf(stderr, "sign_hash err %d: %s\n", err,
            error_to_string(err));
        exit(1);
    }

    wr = fwrite(out, 1, len, stdout);
    if (wr != len) {
        fprintf(stderr, "failed to write key to stdout, %zu/%lu
            bytes written\n", wr, len);
        exit(1);
    }

    ecc_free(&key);
}
```