# Video Services Investigation Inspired by NTU COOL

Survey on the difficulty of video downloading

Kai-En Lin B07902075
Ming-Hsuan Tsai B07902064
Chi-Feng Tsai B07902123
Yu-Heng Chen B07902026

## ABSTRACT

Inspired by the recent controversy over video downloading on *NTU COOL*, we investigated the strategies adopted by different video hosting services to prevent users from downloading videos. After comparing some websites, we proposed 5 different levels of difficulties and did some discussion on our definition. Finally, we recommend a minimum level of difficulty to achieve effective protection.

During the investigation of NTU COOL, we also studied how NTU COOL collected users' watching behavior. We built an automated script as a proof of concept based on our knowledge about the collecting mechanism. We proposed a solution to the operators of NTU COOL, and they said they would adopted it in the future.

## 1 INTRODUCTION

Recently, university courses are forced to become online due to *COVID-19*. Most courses in National Taiwan University are moved to *NTU COOL*. Previously, some students tried to download the course videos on *NTU COOL*, but they are banned since the manager of *NTU COOL* claimed that these videos should only be watched on *NTU COOL*. Downloading the videos is a violation of the lecturer's digital rights.

Currently, to download the videos on *NTU COOL*, one can simply press *F12* and retrieve the URL of the video. Downloading the video is within very few clicks. However, if the lecturer uploaded their videos to other video hosting platforms such as *YouTube* or *U webinar*, we will not be able to find the exact URL of the video. We wonder how these websites protect their videos from being downloaded easily.

In this project, we analyze how popular video hosting platforms make it harder for their users to download the videos. We mainly focus on *YouTube* and *U webinar* since these are the most popular platforms in National Taiwan University. We compare their prevention strategies, find out their respective advantage and disadvantage, and propose a way to classify these methods by their difficulties.

In the previous works [2, 6, 11], researchers often focus on establishing a secure channel from the server to the client's screen, which requires both data encryption and an isolated environment supported by hardware on the client side. However, these works are not easily deployable due to hardware requirements, and thus drive a lot of websites to turn to software-based methods. Though these methods cannot protect videos from being downloaded thoroughly, they can still block the naive users and complicate the development of automated tools to some extent. The goal of this project is to investigate these software-based methods and provide owners of video services with a better view of existing approaches if they want to protect their contents without hardware support.

## 2 PROBLEM DEFINITION

Our goal is to investigate and classify the main factors that increase the difficulty of downloading videos from a web service by surveying several common video hosting platforms, and propose some solutions that efficiently increase the level of protection for contents which are sensitive to reproduction and distribution.

The definition of "downloading" is to obtain the original data delivered to the video player without any forms of destructive conversion or compression. Thus, screen recording is not regarded as downloading because the data has been passed through the rendering of the browser and the system's display server.

We also assume the user who tempted to download videos is not able to directly read the memory in the *Trusted Execution Environment (TEE)* [13] of the hardware, which by definition is a tamper-resistant processing environment that resists against all software attacks as well as the physical attacks performed on the main memory of the system.

## 3 DIFFICULTY RATING

To evaluate the difficulty of downloading videos from websites, we divides the difficulty into five levels. This difficulty rating gives a simpler evaluation of the download prevention scheme of websites.

**Level 1: Users can simply right click on the video and select "save video as"**. This level serves as a lower bound of difficulty, where users can download the video with no effort and automated script is also easy to write. It indicates that the video is played through the `<video>` element with direct link to the static source of the video, and the default listener of `contextmenu` event is not overridden by the site's operator.

**Level 2: The link to the static source of the video is in the "src" attribute of the `<video>` element**. The only difference between *level 1* and *level 2* is whether the default listener of `contextmenu` event is overridden or not. In this level, the convenient menu will not show up when the user right clicks on the video, so a naive user may not be able to download it. However, the link to the video is still in the "`src`" attribute of the `<video>` element, so a more knowledgeable user can still find the link with the help of *developer tools* provided by most mainstream browsers. Note that the difference is only meaningful to human users but not to automated script, since the link to the video is still at the same position in the HTML code.

**Level 3: The URL to the static source of the video exists and direct access is allowed**. In this level, even though there exists a URL that links to the static source of the video, it is not as easy as searching for the `<video>` element directly. The video may be played through *Media Source Extensions API* or higher level protocols like *Dynamic Adaptive Streaming over HTTP (DASH)* [15] or *HTTP Live Streaming (HLS)* [5], where the video stream is delivered by *JavaScript*. However, since the URL is in the page

source, we can find the URL directly without tracing the code, perhaps by observing the the network traffic or just checking every URL in the page. Note that the URL may be in the form of *manifest file* of some protocols which contains multiple links to the segments of the video. To extract the links and reassemble the original file, it may require third-party software. Since most of the well-known protocols are well-documented and have well-developed software, the difficulty of downloading the video is not increased significantly. As a result, we do not classify it as a higher level in our definition.

**Level 4: The video can only be accessed with specific procedure defined in obfuscated or closed-source code**. Generally, if there exists a **necessary** procedure that is not easy to follow or reproduce in the process of delivering the video, we classify it as this level. Note that the procedure must be **necessary**, so if one can query the sources without following the procedure, it is not included in this level. The necessity can be imposed to the user by various methods. For example, the server can require a token that needs to be computed when the client queries the source of the video, or encode/encrypt the data in a form that needs to be processed.

**Level 5: Protected by hardware-implemented *Digital Rights Management (DRM)* technologies**. This level serves as an upper bound of difficulty, where the contents of the video are processed in hardware-implemented *Trusted Execution Environment (TEE)*. With this technology, all the contents are encrypted before being sent. Moreover, the encryption keys are never exposed to normal execution environment and the decrypted contents are also delivered to the screen through a secure channel. Hence, it is impossible to download the video under this level of protection.

Theoretically, it is possible to design a standard procedure to automatically download videos from all websites which are *lower than or equal to level 3*. The procedure would be like scanning all the URLs found in the page source and checking their file formats to apply different methods to different protocol. As for *level 4* and *level 5*, it is necessary to put some efforts to develop a downloading method, if downloading is not impossible. As a result, we consider only difficulty of *level 4 or higher* to be effective protection.

# 4 CASE STUDY

## 4.1 NTU COOL

To download videos from NTU COOL, open chrome's developer tools in Inspect Element mode and find the video element. Since the real location (URL) of the video file is specified in the "`src`" attribute, one can access the video directly with the URL.

An alternating method to download the video is to select the video element and delete event listener that blocks the context menu of the video in the page. By this way, the difficulty is reduced to *level 1*, and one can directly download the video by right clicking the video and select "save video as".

Based on our rating, the difficulty of downloading videos on NTU COOL is *level 2*, since the context menu of the video in the page is blocked. Users cannot download the video via right clicking on the video and selecting "save video as" option. However, it is easy to find the link to the static source of the video using browser's developer tools.

## 4.2 U Webinar

To download videos from U Webinar, one can again use chrome's developer tools and locate the video element. In this case, there is a link of a *M3U8* file. *M3U8* is a manifest file format for a multimedia playlist, and it is the basis for *HTTP Live Streaming (HLS)* [5], which is a communication protocol of deploying content using ordinary web servers and content delivery networks. Once the *M3U8* file is found, one can use video converter tools such as `ffmpeg` to download and assemble the video.

U webinar is an example of *level 3* website in our rating system. Although the URL of the video is rather easy to find, one cannot download the video directly with the URL. Since U webinar uses the *HLS* protocol to distribute audiovisual content, the user needs to recognize *M3U8* manifest file and use additional video converter tools to download videos from U webinar. It is worth noting that U webinar doesn't block the context menu of the video, but since the browser doesn't recognize *M3U8* files as videos, there is no "save video as" option in the context menu.

## 4.3 YouTube

There are several different cases when one tries to download a YouTube video. We make the following assumption to simplify the discussion:

- The video is not age-restricted: This assumption holds in most situations. Most users rarely come across age-restricted videos on YouTube.
- The video is not delivered with DASH protocol [15]: This assumption also holds true in most situations. Although for videos with DASH protocol, the strategy for downloading is practically the same.

To download a video on YouTube, we first go to the page source of the web page. This can be done by right-clicking on the video page and select the appropriate item.

In the page source, we search for the string with the following format: `https:[SERVERID].googlevideo.comvideoplayback?[ARG=VALUE]\\u0026[ARG=VALUE]\\u0026 ...`

The words enclosed in brackets are variables that change across different videos. In most cases, there are multiple such strings. To find the target, we look at the `itag` value that is associated with each string, and is located closely to the string. We show some common `itag` values and their brief descriptions [14]

| `itag` code | container | content | resolution | bitrate |
|---|---|---|---|---|
| 18 | mp4 | audio/video | 360p | - |
| 22 | mp4 | audio/video | 720p | - |
| 137 | mp4 | video | 1080p | - |
| 140 | m4a | audio | - | 128k |

Once we find the desirable string, we modify the string to get the URL of the stream itself[8].

- Change all "`\/`" to "`/`"
- Change all "`\\u0026`" to "`&`"

In most cases, the URL is sufficient to download the stream. However, for some videos, especially those uploaded by verified channels, the URL may lead to a 403 error. This is due to the lack of

signature in the URL. In this scenario, we can find the "signatureCipher" attribute near the string followed by `s=...`, where `...` is the ciphered signature.

To decipher the signature, we need the functions implemented in the *Javascript* code named `base.js`, which can be found in the page source.

One can easily recognizes `base.js` as an obfuscated source code, since there are thousands of functions with meaningless function names. To identify the function that is related to the signature, which we later refer to as the "initial function", we make use of regular expressions. Based on our research, observations, and experiments, we came to the following conclusion about the initial function.

- The function name usually consists of two alphabets, with the first one capitalized, e.g. `Pu`.
- It usually takes only one parameter e.g. `Pu=function(a)`
- It works by applying several other functions with similar function names sequentially, e.g. `Pu=function(a){a=a.split("");Ou.Kt(a,78);Ou.uu(a,43)...}`

In fact, we have found some specific regular expressions used by the tool "pytube"[12] to identify the function. However, there are more than ten possible forms, and each of them is very complicated. Based on our own observation above, we can successfully locate the function in most situations. Continuing with the examples, the remaining task is to find the functions such as `Ou.Kt`. However, the prefix `Ou.` is an obfuscated variable name, which means that instead of looking for `Ou.Kt`, we should be looking for `Kt`. Once we find all the functions used by the initial function, we can reproduce their functionalities and decipher the signature[1]. For brevity, we have omitted the details about how we make use of these functions to retrieve the signature.

With the signature, we can add it to the URL as an argument, e.g. `&sig=...`. In this way, we can successfully access the video.

Based on the discussion above, we classify *YouTube* as *level 4*. Although the information needed to retrieve the URL of the video, including deciphering the signature, can all be found in the page source, the information cannot not be easily obtained. Even for videos that do not require deciphering the signature, the URLs are not placed under common HTML tags, and they are not in the form of an usual URL, so they are not easily identified by users.

### 4.4 Netflix

Different from previously discussed platforms, *Netflix* is a paid service. It is necessary for them to provide robust protection against unauthorized access, which is a great threat to their profits. Currently, *Widevine* framework [6], a DRM solution that supports 3 different levels of protection, is used in their website and applications. The most secure level is *L1*, where all operations are handled inside a *TEE*, whereas the least secure level is *L3*, which is implemented without hardware support. In *Netflix*, only high quality videos are protected by *L1* and the others are protected by *L3*. However, the difficulty of cracking *L3* is still greater than all the cases discussed previously because the data is stored in encrypted format. The attacker can only intercept the plain contents by side-channel attack (e.g. inspect the memory) or cryptographically breaking it. Based on these facts, we classify *Netflix* as *level 5* for videos protected by *Widevine L1* and *level 4* for videos protected by *Widevine L3*.

Although both *YouTube*'s scheme and *Widevine L3* are classified as *level 4*, we can see that *Widevine L3* is much more stronger. The key difference is that *Widevine* uses the browser's closed-source module to handle cryptographic operations, which is hard for reverse engineering. On the other hand, *YouTube*'s code is in *JavaScript* and directly exposed to the client, so it is relatively easy to trace even if it is obfuscated.

[1]We were interested in the content of the signature. For example, whether it was the hash value of certain properties of the user and the video. However, neither ciphered nor plain-text signature was human-readable message, so, unfortunately, we were unable to interpret the content of the signature.

## 5 RECOMMENDED SOLUTION

To fully protect a video from being downloaded, a hardware solution may be inevitable. Regardless of the encryption method used during the video transfer, the plain content must be loaded to memory in order to render the video on the screen. Once the video is loaded into memory, it is completely at the client's disposal [16].

However, stronger protection often leads to more overheads and the requirement for hardware may reduce the usability of a product, so a trade-off must be made. Moreover, allowing the download of videos may not be a vital problem that impairs a service's profit. Thus, the strongest solution may not always be the best choice. For example, *Widevine L3* outperforms *Widevine L1* in terms of usability, while simple obfuscation scheme like *YouTube*'s signature outperforms *Widevine L3* in terms of efficiency.

The solution depends on the service provider's policy. Generally, for a service hosting contents with copyright, we recommend that the difficulty of downloading should be at least *level 4* to ensure a minimum effort from the client to bypass the protection. If one needs to offer a strong protection that is hard to crack by any existing strategy and the overhead of cryptographic operations is not a critical issue, then it should seek a mature *level 5* solution like *Widevine L1*, combined with a stronger *level 4* solution like *Widevine L3* to extend its usability. Otherwise, if it needs to strike a balance between strength and efficiency, then it should use a light-weight scheme to achieve the property of *level 4* like what *YouTube* uses.

## 6 RELATED WORK

The *Dynamic Adaptive Streaming over HTTP (DASH)* [15] technique, which is commonly used by servers [1, 10], can act as a basic protection that prevents a user from downloading the videos directly. Similar to other streaming protocols, a video is partitioned into segments and each of them is delivered independently. This way, the client does not receive the entire video as a file that can be saved directly. However, one can still investigate the implementation of the protocol, and develop a tool that captures the segments and assembles them to retrieve the original file.

The idea of video encryption gives a better solution for the problem. Some studies focus on combining media streaming protocols with existing *DRM* solutions to address the problem. Yuanzhi Zou et al. propose an H.264 video content encryption scheme [18], which can be applied to *DRM* for mobile video, download, live-streaming

device, etc. Hartung et al. proposed changes and extensions to *DASH* [7] in order to incorporate *DRM*. Later, *W3C* introduced *Encrypted Media Extensions (EME)* [4] as an common API that connects *Media Source Extensions (MSE)* [3] with any DRM systems or simpler key management systems without the use of heavy third-party plugins. Many huge companies then developed their own solutions based on *EME*, such as Google's *Widevine* [6], Microsoft's *PlayReady* [11], and Apple's *FairPlay* [2].

To achieve the properties of *DRM*, it relies on the concept of *Trusted Computing* to achieve secure computation, privacy and data protection. Sabt et al. gives a general definition of a *Trusted Execution Environment* [13], while previous studies, such as *SeCReT* [9], proposed a framework that strengthens the security of the communication channel between *TEE* and normal execution environment.

In the investigation of *YouTube*, we rely on the tool *pytube* [12], a lightweight *Python* library for downloading *YouTube* Videos. The main process of the investigation is based on understanding and tracing the source code of *pytube*. Another well-known tool for downloading videos is *youtube-dl* [17], which allows users to download videos from *YouTube* and supports many other video platforms. However, the structure of its code is extremely complex thanks to its various downloading options and supported platforms, and this is why we didn't use it as our main investigating tool.

## 7 CONCLUSION

While video-hosting becomes a common service on the Internet, it is important for the service provider to protect the author's original contents from being downloaded and distributed. Though some *DRM* technologies have been developed to meet this demand, they are not widely deployed due to their high costs. As a result, many websites turn to techniques that only increase the difficulty. We studied these techniques by surveying several websites and categorized them for the purpose of comparison and evaluation.

We proposed a difficulty rating with 5 levels to simplify the process of evaluating the strength of a download prevention scheme and concluded a minimum level to form effective protection. We investigated the video-streaming mechanism of several platforms (*NTU COOL, U webinar, YouTube,* and *Netflix*) and classify their difficulty levels. At last, we proposed some directions to help web developers to increase the effort for malicious downloaders or even eliminate the chance of unauthorized video downloading. Implementing those proposals could improve security and ensure the Digital rights of video vendors.

## 8 FUTURE WORK

As mentioned before, *video encryption* is the only ultimate solution to prevent unauthorized video downloading. However, the overhead of encryption and decryption is a main reason of its low adoption rate on free streaming content. As a future work, we would like to investigate and quantify the influence of video encryption. Moreover, we want to focus on improving the performance of video encryption.

In the future we also plan to write an automatic script that can access video platforms and determine the difficulty of downloading videos from them. With this program, we can implement a general survey on those top video-streaming and live-streaming platforms, and find out the proportion of insecure (difficulty *lower than or equal to level 3*) websites.

## REFERENCES


[1] Javier Añorga, Saioa Arrizabalaga, Beatriz Sedano, Maykel Alonso-Arce, and Jaizki Mendizabal. 2015. *YouTube's DASH implementation analysis.*
[2] Apple. [n.d.]. *FairPlay.* https://developer.apple.com/streaming/fps/
[3] Aaron Colwell, Adrian Bateman, and Mark Watson. 2014. Media source extensions. (2014). https://www.w3.org/TR/media-source/
[4] D Dorwin, A Bateman, and M Watson. 2015. W3c editor's draft: Encrypted media extensions. https://w3c.github.io/encrypted-media
[5] Andrew Fecheyr-Lippens. 2010. *A review of http live streaming.*
[6] Google. [n.d.]. *Widevine.* https://www.widevine.com/
[7] Frank Hartung. 2011. *DRM protected dynamic adaptive HTTP streaming.* ACM SIGMM Conference on Multimedia Systems, MMSys.
[8] Dr. Mark Humphrys. [n.d.]. *How to write a Shell script to download videos from YouTube.* School of Computing. Dublin City University. https://www.computing.dcu.ie/~humphrys/Notes/UNIX/lab.youtube.html?fbclid=IwAR07tFK4Nq6gVSIhemCWp2K9W91dEW_LXjKeEMvgkqpXSn3ZFb9Dx5EMEcw
[9] Jinsoo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Kang. 2015. *SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment.* https://doi.org/10.14722/ndss.2015.23189
[10] D. K. Krishnappa, D. Bhat, and M. Zink. 2013. *DASHing YouTube: An analysis of using DASH in YouTube video service.*
[11] Microsoft. [n.d.]. *PlayReady.* https://www.microsoft.com/playready/
[12] nficano. [n.d.]. *pytube.* https://github.com/nficano/pytube
[13] M. Sabt, M. Achemlal, and A. Bouabdallah. 2015. *Trusted Execution Environment: What It is, and What It is Not.*
[14] sidneys. [n.d.]. *YouTube-itag-list.* https://gist.github.com/sidneys/7095afe4da4ae58694d128b1034e01e2
[15] Thomas Stockhammer. 2011. *Dynamic adaptive streaming over HTTP: Standards and design principles.* IEEE International Conference on Multimedia Systems.
[16] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2013. *Steal this movie - automatically bypassing DRM protection in streaming media services.* the 22nd USENIX conference on Security.
[17] ytdl org. [n.d.]. *youtube-dl.* ytdl-org. https://github.com/ytdl-org/youtube-dl
[18] Y. Zou, T. Huang, W. Gao, and L. Huo. 2006. *H.264 video encryption scheme adaptive to DRM.*


## A SIDE PROJECT - *NTU COOL* VIDEO AUTO-WATCHER

### A.1 Motivation

As mentioned in the introduction, the operator of *NTU COOL* banned the students from downloading the videos. Nevertheless, some students still tended to download videos due to the lack of convenience and functionality of *NTU COOL*'s web interface.

Here comes the problem. *NTU COOL* will gather our video watching records if we watch videos on *NTU COOL*'s website. If we download the video and watch it, we will not have the records, which is used by some lecturers to decide the participation grade of students. Some students have tried to play the video on *NTU COOL* in background process to obtain the watching records, but sometimes this method does not work.

In this side project, we performed a black-box testing on *NTU COOL*'s video module. We analyzed the network packets between our browser and *NTU COOL*'s server and figured out how the server collects the information about our watching behavior. We found the defect in the information collecting and built a script to automatically send watching records by exploiting the defect.

### A.2 Result

We used `Google Chrome 83.0.4103.116` to perform our experiment. We navigated to a page which contained a video. Then, we

opened Chrome DevTools and selected network panel. We played, stopped, and changed the quality of the video, and observed the network packets corresponding to our actions.

#### *A.2.1 Interesting Requests.*

We found the following two requests which contained some interesting information:

- `video_last_plays`
  The content of this request is a *JSON* like `{last_play_time: 83, video_id: 16007}`. The name `last_play_time` is the current timestamp of the video (in seconds).
  This request will be sent every 30 seconds during the playing of the video. Stopping, changing playback rate, changing quality, leaving the page, or reaching the end of the video will trigger this request as well.
  After some testing, we confirmed that this request affected the starting time of our next visiting.

- `video_viewing_records`
  The content of this request is a *JSON* like `{playback_rate: 1, start: 80, end: 82, video_id: 16007}`. The server will respond us with another JSON like `{id: 61017495, start: 80, end: 82, student_id: 8655, course_id: 1198, video_id: 16007, created_at: "2020-07-05T13: 39: 26.399Z", updated_at: "2020-07-05T13: 39: 26.399Z", playback_rate: 1.0}`.
  This request will be sent when we stop, change playback rate, change quality, leave the page, or reach the end of the video.
  After some testing, we confirmed that the server used this request to gather the information of our watching. As we can see in the responding packet, the server will record video playback rate, the starting and ending timestamp of our watching, and the time when we watch the video.

#### *A.2.2 Auto-watching Script.*

In the previous subsection, we have showed two requests related to video playing. `video_viewing_records` is the request through which the server collects our watching record. We built a *Python* script that could log in to *NTU COOL*, scan for all video links in a given course, and send `video_viewing_records` to forge watching records. To make the fake watching records more persuasive, we add some randomness into the watching behavior. More precisely, the script sends watching records in the following strategies: ($L$ means the length of the video)

- watch from 0 second to $L$ second
- randomly and uniformly select two integers from 0 to $L$. Let them be $x$ and $y$. Watch from $\min(x, y)$ second to $\max(x, y)$ second
- at 50% chance, watch from 0 second to $\frac{1}{3}L$ second
- at 50% chance, watch from $\frac{2}{3}L$ second to $L$ second
- at 33% chance, simulate the user fast-forwarding the video a few times by sending discontinuous watching records

Note that during the development of this script, we also found that querying `https://lti.dlc.ntu.edu.tw/api/v1/courses/{course_id}/videos/{video_id}` led to a *JSON* containing detailed information about the video. URL to the video of different resolution, the last updated time, teacher's ID, etc., can be found in this *JSON*. If we want to download the video with the highest quality, we can find the URL to it in this *JSON*.

#### *A.2.3 Server-side Countermeasure.*

Inspired by *SYN cookie*, we came up with a protocol to mitigate the forging of the watching record.

(1) The user requests for the video.
(2) The server responds the video, user ID $U$, video ID $V$, current time $T_0$, and a message authentication code $\text{MAC}_{pk}(U, V, T_0)$. The MAC is signed with the server's private key $pk$.
(3) After watching the video, the user sends a normal `video_viewing_records` to the server with the MAC mentioned above.
(4) The server verifies the integrity of $U$, $V$, and $T_0$ using MAC. Then, it calculates the passing time $\Delta T = T_1 - T_0$, where $T_1$ is the time when receiving `video_viewing_records`. Let $s, e, p$ be the `start`, `end`, and `playback_rate` in `video_viewing_records`, respectively. If $\frac{(e-s)}{p} > \Delta T$, the server rejects the request.

This protocol is resistant to some replay attack since the MAC includes user ID and video ID.

The protocol prevents users from, for example, claiming he watched a 2-hour video just 2 minutes after receiving the video. Nevertheless, an intended user can still do things like receiving the MAC and send a `video_viewing_records` 2 hours afterwards. We do not take this kind of fraud into consideration since it takes much more effort to perform such fraud.

#### *A.2.4 Report the Problem.*

We reported the our findings and provided our protocol to the operators of *NTU COOL*. They confirmed that the system was vulnerable to fake viewing records. They will update the video module using our protocol in the future, and will inform the teachers that viewing counts may potentially be fake.

## A.3 Conclusion

In this side project, we investigate how *NTU COOL* collects our video watching records. We found that the mechanism was too simple to hack, and therefore we built a Python script to send fake watching records as a proof of concept. In the end, we reported the problem to the operators of *NTU COOL* and proposed the solution protocol designed by ourselves. They adopted our proposal and would update the video module in the future.