International Journal on Semantic Web and Information Systems
Volume 20 • Issue 1

Digital Copyright Management Mechanism Based on Dynamic Encryption for Multiplatform Browsers

Ming-Te Chen, National Chin-Yi University of Technology, Taiwan*
https://orcid.org/0000-0001-9583-4419

Yi Yang Chang, National Chin-Yi University of Technology, Taiwan

Ta Jen Wu, National Chin-Yi University of Technology, Taiwan
https://orcid.org/0009-0003-6731-8187

ABSTRACT

In recent years, the internet and smart devices have developed rapidly. Many people no longer rely on newspapers, magazines, or television to receive news. They can see the latest news using computers or mobile phones. According to a study by the Taiwan Internet Information Center, nearly 90% of Taiwanese people have used the internet. Many online streaming services have emerged, and people can easily watch movies and TV programs through computers or mobile phones. Hence, some websites use digital copyright management mechanisms to protect videos from being directly downloaded. However, 30% of websites use AES-128 encryption to protect their content. If the key access mechanism is not well protected, the encryption methodology may be useless. Therefore, this paper proposes a cross-platform digital copyright management mechanism for adaptive streaming. With this mechanism, users do not need to download additional applications, as the mechanism implements Web-Assembly language through the browser.

KEYWORDS

Adaptive Streaming, Digital Right Management, Dynamic Encryption, RSA Encryption, Web-Assembly Language

INTRODUCTION

With the advancement of technology and the popularization of the internet, many people have begun chatting or watching news or other videos online. Figure 1 depicts the 2019 Taiwan Internet Report released by the Taiwan Internet Information Center (Taiwan Network Information Center, 2019). According to this report, 89.6% of the domestic population older than 12 has used the internet, and 85.6% have used audio-visual and live broadcasts. This report also demonstrates that most people watch videos through the internet. Modern people no longer obtain news and entertainment through

DOI: 10.4018/IJSWIS.334591
*Corresponding Author

only television programs or newspapers and magazines, and they do not need to stay in front of the TV or go to the cinema to watch movies and TV series. Modern people are adopting different electronic products, such as computers, tablets, mobile phones or TV boxes, to obtain information, watch videos or listen to music.

Various audio and video streaming formats such as Moving Picture Experts Group-Dynamic Adaptive Streaming over HTTP (MPEG_DASH) and HTTP Live Streaming (HLS), as well as digital rights management systems such as Widevine and FairPlay, have been developed to facilitate the smooth transmission of audio and video content over the internet. Many movie production companies and TV stations have started to provide users with online platforms to stream movies or music and have developed video streaming platforms, such as HBO Max and Disney+, which use digital rights management to mitigate the spread of piracy and directly downloading movies. However, most user's device has the limitation on the device such as the mobile phone or tablet. Some of them are equipment with limited memory space and lower computation power in their above devices. On the other hand, the streaming media format does not have the standard such as the HLS format for Apple's device, another is MPEG-DASH, RTMP, and the last on is the SMOOTH format. In order to provide services to users across various devices, these streaming services must convert videos into various formats, which is a large burden on storage space. Furthermore, different DRMs support only specific browsers or operating systems. To protect video and audio content, 44.8% of platforms use paid third-party DRM solutions. As shown in Figure 2, 40% of video and audio platforms still do not use any digital rights management systems. 29% of platforms use HLS+AES-128; however, this system may not have any effect if the key acquisition method is not controlled.

Therefore, this paper uses (Wu, 2020) as a foundation to propose building a streaming digital rights management mechanism that supports multiplatform browsers and stores only one file format with the dynamic encryption of video clips. Not only this approach can let users prevent installing extra software or browser's plugins to fetch the desired video clip efficiently but also it also reduces the redundant encrypted media file format to decrypt by the video server.

Figure 1. Estimated number of internet users in Taiwan in 2019

Source: 2019 Taiwan internet Report

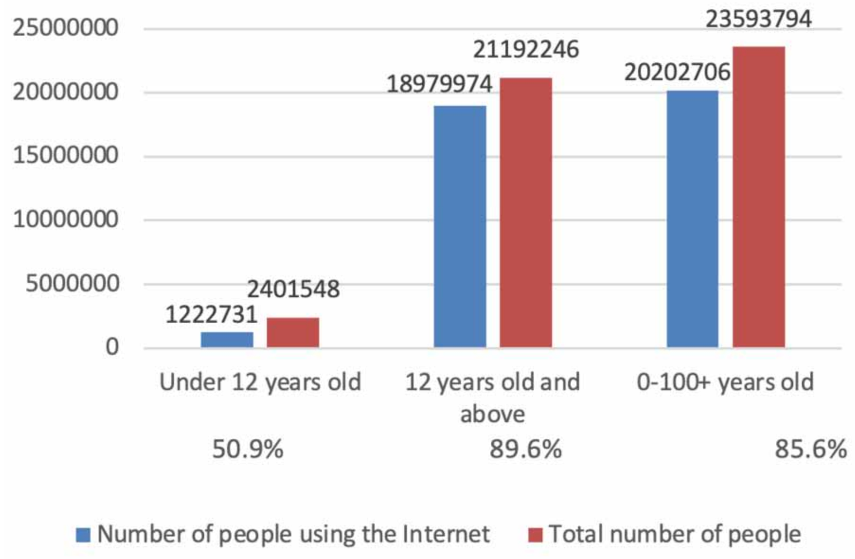
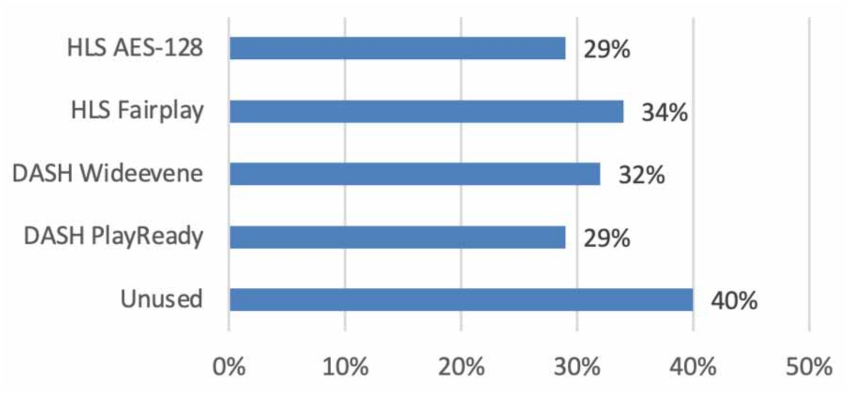


Figure 2. Statistics on the use of encryption methods in the industry
Source: Bitmovin 2019 Developer Report



RELATED WORKS

In recently years, there are some articles to discuss the digital right management methods such as the Attrapadung et al., (2018) proposed their scheme which their method could implement two level homomorphic public-key encryption with bilinear mapping group by adopting WebAssembly. Their approach does not have any other plugins and could execute on the web-browser efficiently. However, their scheme also needs higher computation cost due to the applying bilinear-mapping operation with homomorphic encryption. By the way, their scheme will increase the cipher-text size when applying on the second level encryption under the addition operations. Song et al., (2023) proposed a new data sharing mechanism called COAB-PRE that it could outsource the attribute encryption to other nodes, and it also implements by adopting WebAssembly. Their approach does not state the video size limitation and brings lots of burden to the edge devices or computation power limitation equipments. Cabrera et al., (2021) proposed a CROW tool that it could apply to the libsodium library and combine the libsodium into WebAssembly together. The CROW tool only could divide the LLVM IR code but not on the WebAssembly code directly. This causes the CROW limitation on the WebAssembly code. We thought that if it could divide the WebAssembly code directly without transferring the code to LLVM IR type. It will cause lots of help to WebAssembly diversification more. Sun et al., (2019) proposed their method that it adopts WebAssembly to design a code protection mechanism called SELWasm for offering encrypting Wasm executing code by applying the cryptographic algorithms. But their scheme also has the whitelist to check which code needs to be encrypted or not. This whitelist's security protection is a serious problem. Meanwhile, this protection mechanism also brings the code to expand its own size. It also increases the decryption time especially when applying AES on the large media files to performing decryption. Sun et al., (2020) showed their attribute encryption method called CP-AB-KEM that it also adopts WebAssembly and Crypto library to perform files access-control. It provides the data privacy protection, users key revocation, and other features. But it also has to build up its own cloud servers and it cannot apply on the computation power limitation resources. By the way, the cloud server's trust level assumption has also to be considered in their scheme.

All above discuss the mechanism that adopting WebAssembly to design the application or new encryption/decryption methods. Some of them must use the asymmetric encryption method or complicated encryption method to design the digital right management application. In one hand, it also fits to specific situation and brings large burden to users. In common cases, most users prefer to use light-weight software or plugins to fetch media stream clips. Hence, we thought that if there is a light-weight method without further installing extra software or plugins. We found that the digital rights management method proposed by Hassan et al., (2020) is based on AES, elliptic curve cryptography (ECC), and other methods. These methods include 256 encryption, user authentication, key transfer and decryption through the desktop application, and digital signature and shared key through ECC-256. Although this method improves that encryption strength of AES-256, the current streaming standard still uses AES-128. However, users must download additional applications to watch videos, which may degrade the user experience. Hence, it may still result lots of decryption time to users when they download large video stream clips from the web site. By the way, users must install the dedicated software to download the desired video clip from the web site after they have authentication with server completely. Therefore, based on the analysis of current methods and related research, this paper improves the security of AES-128 and proposes a digital rights management approach that complies with the streaming standard and can be executed directly through a browser without the need to install additional programs to protect the key efficiently.

EXPERIMENTAL METHODS

1. **Experimental Architecture:**
2. In this paper, the experiment has two main parts, namely, the server side and the user side. The server side includes the authentication server, the audio/video server and the key server, and the user side executes JavaScript and WebAssembly through the browser of a computer, cell phone or tablet. The overall experimental architecture is shown in Figure 3.

The server side is divided into three parts, as shown in Figure 4. The authentication server handles user authentication and key issuance, the video server delivers video playlists and clips, and the key server delivers clip encryption keys. The user side is divided into two parts, as shown in Figure 5. JavaScript handles the video player, and WebAssembly sends authentication, play URL and key requests to each server.

The front-end part is controlled by JavaScript, and the general streaming request is executed through JavaScript. In this paper, WebAssembly controls user authentication, video URL requests and key requests and decryption. On the back-end, the authentication server is responsible for confirming user privileges and then allocating a set of tokens for subsequent calls to other APIs. The server will also provide an RSA public key to use when requesting fragment keys later. After obtaining the token, the user can request the video playlist from the video server and send it to the player; when the player is ready to play, it obtains the video encryption key from the key server through WebAssembly key encryption and then provides the RSA public key generated on the user side to the server. The key server decrypts the request with its own private key and confirms that the request is valid, encrypts it with the public key of the user side, and sends it back to the user side. This request is then sent to

Figure 3. Experimental architecture

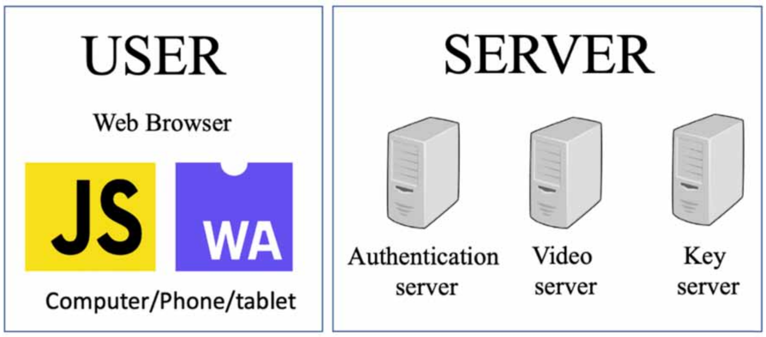


Figure 4. Server architecture

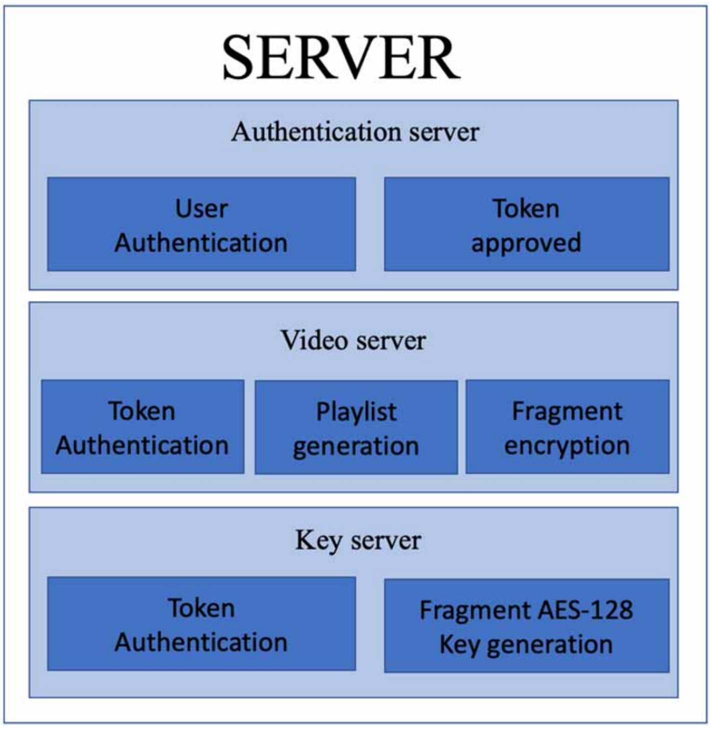
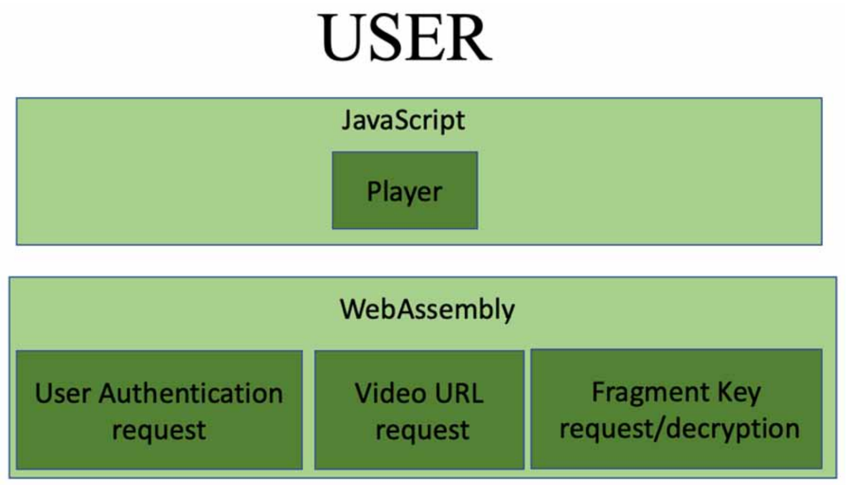
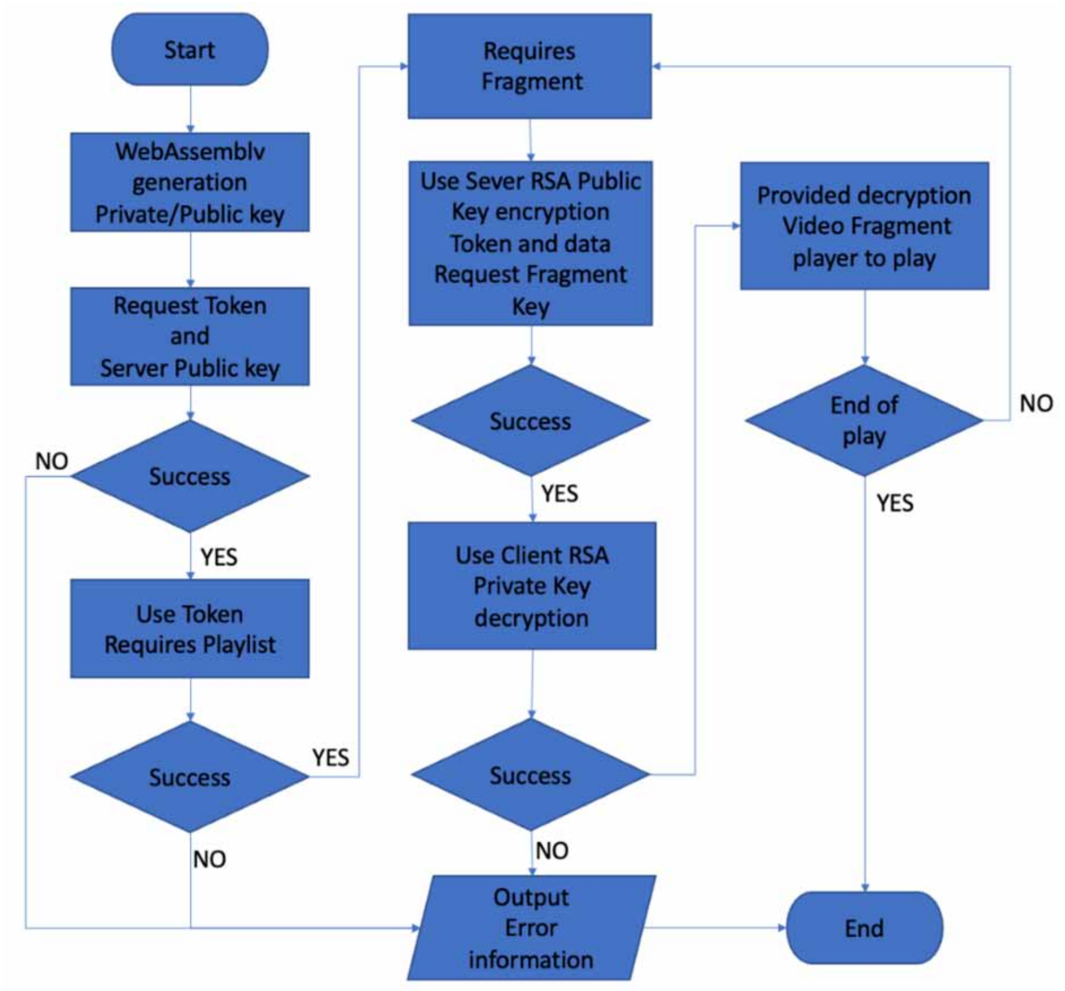


Figure 5. Client architecture



the user throu gh WebAssembly decrypting with the private key of the user and finally passed to the player for playback. The overall playback process is shown in Figure 6, and the overall request flow is shown in Figure 7.-

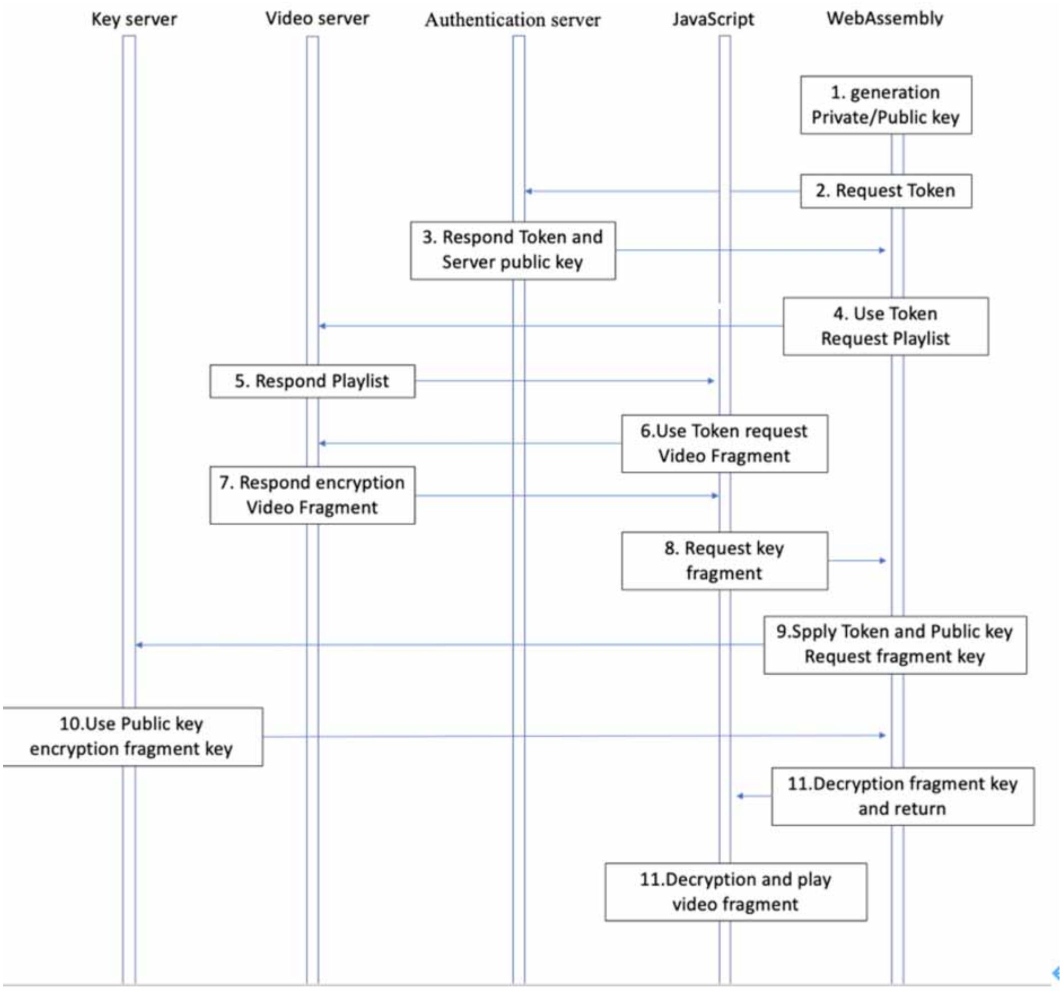
Figure 6. Playback flow chart


Start
WebAssemblv generation Private/Public key
Request Token and Server Public key
Success
NO
YES
Use Token Requires Playlist
Success
YES
NO
Requires Fragment
Use Sever RSA Public Key encryption Token and data Request Fragment Key
Success
YES
Use Client RSA Private Key decryption
Success
NO
Output Error information
Provided decryption Video Fragment player to play
End of play
NO
YES
End

EXPERIMENTAL RESULTS

1. **Experimental Environment:** The video used is from Big Buck Bunny and uses ffmpeg to convert the video to 1920x1080, 1280x720, 720x480, and 640x360. The bit rates are 3000k, 2400k, 1400k, and 800k. The video then uses Bento4's mp4-hls.py to convert to unencrypted HLS format and divide the video into 5 second segments, with 106 segments in total. The authentication server and key server use nginx and PHP, the video server uses NodeJS's Express, and all servers use the HTTPS protocol to encrypt the connection. The server CPU is Intel i7-7700. The front-end player uses Video.js and modifies the key request process, while the WebAssembly portion is written and compiled in Go.
2. **Experimental Results**: When WebAssembly is initialized, a set of RSA 2048-bit private and public keys are generated. The private key is only stored internally and used to decrypt the key data returned by the server. The public key is provided to the server to generate the result, as shown in Figure 8.

Figure 7. Authentication and playback flow chart


Key server
Video server
Authentication server
JavaScript
WebAssembly
1. generation Private/Public key
2. Request Token
3. Respond Token and Server public key
4. Use Token Request Playlist
5. Respond Playlist
6.Use Token request Video Fragment
7. Respond encryption Video Fragment
8. Request key fragment
9.Spply Token and Public key Request fragment key
10.Use Public key encryption fragment key
11.Decryption fragment key and return
11.Decryption and play video fragment

Before playing the video, a set of tokens is obtained from the server. The token corresponds to the ID of the user, the video to be played and the public key provided by the server. The successful request results are shown in Figure 9, and the original data of the token is shown in Figure 10.

After the token is obtained, the playlist URL can be obtained from the server. This request result is shown in Figure 11.

The player then requests the playlist. The main playlist is shown in Figure 12, and the clip playlist is shown in Figure 13.

During playback time, the player finds an EXT-X-KEY tag, which means that the content is encrypted, so it requests the URL in the tag. Here, through the modified player request process, the part after key is provided to the key and the server makes a request. The request results and the unencrypted original data are shown in Figure 14. The original data of the key request is shown in Figure 15, and the original data of the key response is shown in Figure 16.

3. **Cross Platform Testing:** After completing the playback processing program, the next step is to test whether it can be executed normally on various browsers in commonly used operating

Figure 8. WebAssembly RSA public and private key generation results

RSA Private Key & Public Key Generate by WASM

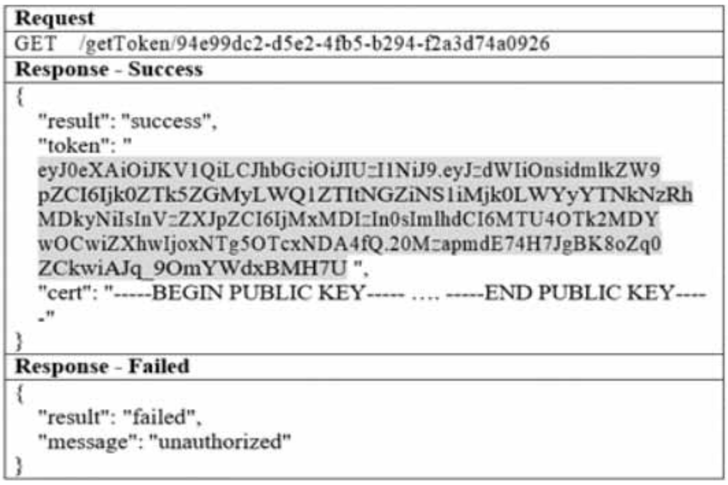
```
-----BEGIN PRIVATE KEY-----
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQDjy5KD3KzItLYS
Nw6d/YOiiMO5zpsa4Oq5EzMZOWdy4GpiSws60QN7ce        NP8ZXF2rielswTDYuc2nPu
52G4iy+kMjcVLBJczvuXuJMUyVLR9i5+MwRYQVxbFqF93plgq+VMWYtj8VuyuVPn
uiJfHQids10v2oCbPUM3u2TX9XTtXDZVZO1zNpHCd6m/QKCBHB3345u0HcMQxeMq
5TlU8XRgYOckrcEaGm/6NEWNjTi4Xnzvwpk4Z1DmoMyqgPTd7pvSt0Br9zHDDuyI
tMhTGL7NHy5PJMvzed/5+Nfb++oBm7kOqP6awY+W3rJo6zhlZO20vnBIbQJlHtx+
J+sJ46RtAgMBAAECggEAY+GfHLJmEJssOlBp2FyQm3PcXlx5Jg/4263Pef/xnC1K
eE7v XZrILBi D00naoiaFTQRR9wi5bs1aiCZscr6U5SoffOk WJBudtaj41ORxrB6F
xFAaV5hRJp+Fr+sq1XiJ0jBxLiFmU7dvUuwKpinU3VEISzNDsM3r60P/s7Gs02YH
pyGXj9NXIFxrxA8xP/T77seMQtAetk2guWWIyI8sYZOEpe0TG7q3MNF3kK9naBaT
qovz21CAXUvQvVzPi5viaB4WZ8BE2/3nTl7lPDYaZKx6LK/98KvmtieVMMkCPaJA
zUJ1uvqZktK5SBR4dcVxrCW2xH4d+ET6csNbXZrIAQKBgQDw8hAtkHfldsaEaTTZ
T03PvUOzlA7JTTt1ejjujwhLeGpNX3UMTnYe7vW8PrRKHe84ZZyNy3bIJNXZMpNN
t3EOJxdxDnZYJeV7EOyVhv1MiMK3IIzY+0WErYJQVDtp9BlGzh9ZOt4A81PVX4Hy
uCuaW9mEDymoOerC0f1cjKr0yQKBgQDyBysjyvx5F1k72l35zbAfuk79IPIrT2dx
nl9NNpaV8RYwWZbYERPmCXTpXC8YleQwAQeONipxivMFobbNj5UXCtX87PsJQVeY
swXiu+Y4cVehJjxYKpNdRqJUkb7nwxBt1nWp9WjKYrheOS6KkwPzHoetIWpizobk
JsTxZxK4hQKBgAQ5ajkySDfYrkBSC+AneMjEm7RBS0USOpGjs+3SyiVwlgSHQOkX
UxDVxa8oNkb6YWBOv8DoqUt3AdzN+zEony3NV6bexwpil7RhppQ0iqpV13wYc      P+4
4A4kcJ1x7Jy979eC85+/3XaMzZJ3GzMjMXiCVbzv+x387fW+cr+1o6WpAoGBALV7
NpfyP3q1TU4aIdmMqlJB+7THHY/oTJLfxGprgmMQkijTJM8Djz4w3R+W5iGzU5XD
T90NNKpLHIuehSqCfbL3hxnOwTZhbsOricBIUQsFwkWuHBWoZzPMITD00IeMtycF
Ztl4kYkgmUrd6c4zCwUkYWwq4FESuAcgIxE4zsrZAoGBAKGtjqMgchkyl  z3SzvIl
2VDIejEeK3CNvUOBYDwJBYtv5dLBPYVNliRfCNpSy/10q+7EEWZoDn7JTJEkG1LT
q/NoPFMPyDjr+f6ZYFXADidAi5KxxjMHrRDAFoBy0F/sX20cSkT/e68DsavWBjDo
CC0MGf2Q45XPOMhg0lXorzqY
-----END PRIVATE KEY-----

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA48uSg9ysyLS2EjcOnf2D
oojDuc6bGuDquRMzGTlncuBqYksLOtEDe3HjT/GVxdq4npbMEw2LnNpz7udhuIsv
pDI3FSwSXM77l7iTFMlS0fYufjMEWEFcWxahfd6ZYKvlTFmLY/FbsrlT57oiXx0I
nbNdL9qAmz1DN7tk1/V07Vw2VWTtczaRwnepv0CggRwd9+ObtB3DEMXjKuU5VPF0
YGDnJK3BGhpv+jRFjY04uF5878KZOGdQ5qDMqoD03e6b0rdAa/cxww7siLTIUxi+
zR8uTyTL83nf+fjX2/  vqAZu5Dqj+msGPlt6yaOs4ZWTttL5wSG0CZR7cfifrCeOk
bQIDAQAB
-----END PUBLIC KEY-----
```

systems. Figure 17 shows the operating system and browser versions used in the test and the results of the test. Figures 18-23 show the playback status of each browser.

Analysis and Comparison

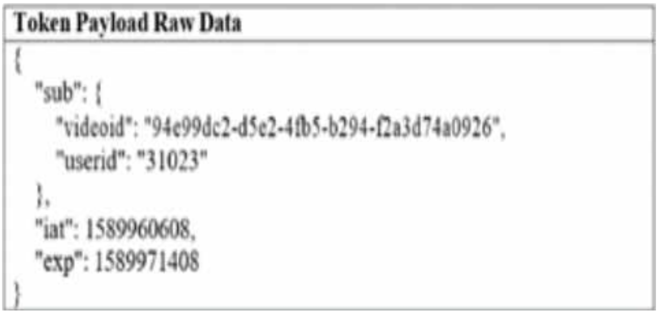
a. Result Analysis: Figure 24 shows the time it takes for the key server to decrypt the encrypted data sent from the client and the time it takes for the server to encrypt the encrypted data with the client’s public key. Server decryption takes approximately 1 millisecond to 1.5 milliseconds, and server encryption with a public key takes 0.04 to 0.1 milliseconds. Figure 25 shows the time

Figure 9. Token request results



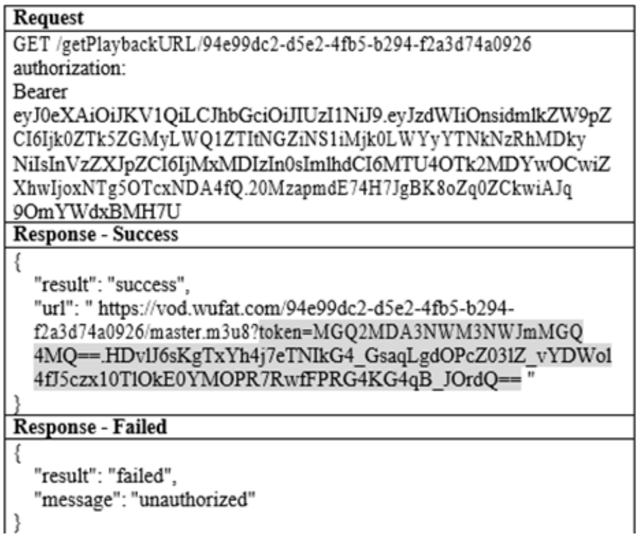
```
Request
GET   /getToken/94e99dc2-d5e2-4fb5-b294-f2a3d74a0926
Response - Success
{
   "result": "success",
   "token": "
   eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOnsidmlkZW9
   pZCI6Ijk0ZTk5ZGMyLWQ1ZTItNGZiNS1iMjk0LWYyYTNkNzRh
   MDkyNiIsInVzZXJpZCI6IjMxMDIzIn0sImlhdCI6MTU4OTk2MDY
   wOCwiZXhwIjoxNTg5OTcxNDA4fQ.20MzapmdE74H7JgBK8oZq0
   ZCkwiAJq_9OmYWdxBMH7U ",
   "cert": "-----BEGIN PUBLIC KEY----- …. -----END PUBLIC KEY----
   -"
}
Response - Failed
{
   "result": "failed",
   "message": "unauthorized"
}
```

Figure 10. Token raw data



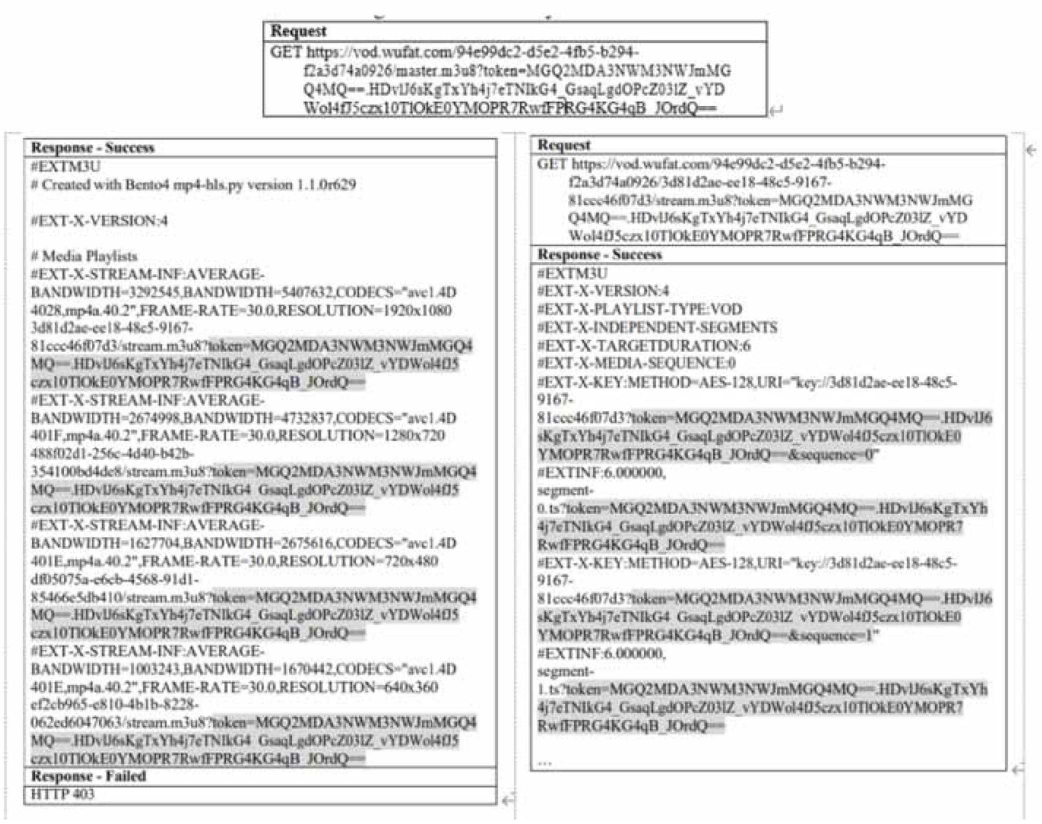
```
Token Payload Raw Data
{
   "sub": {
      "videoid": "94e99dc2-d5e2-4fb5-b294-f2a3d74a0926",
      "userid": "31023"
   },
   "iat": 1589960608,
   "exp": 1589971408
}
```

Figure 11. Playlist URL request results



```
Request
GET /getPlaybackURL/94e99dc2-d5e2-4fb5-b294-f2a3d74a0926
authorization:
Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOnsidmlkZW9pZ
CI6Ijk0ZTk5ZGMyLWQ1ZTItNGZiNS1iMjk0LWYyYTNkNzRhMDky
NiIsInVzZXJpZCI6IjMxMDIzIn0sImlhdCI6MTU4OTk2MDYwOCwiZ
XhwIjoxNTg5OTcxNDA4fQ.20MzapmdE74H7JgBK8oZq0ZCkwiAJq
9OmYWdxBMH7U
Response - Success
{
   "result": "success",
   "url": " https://vod.wufat.com/94e99dc2-d5e2-4fb5-b294-
   f2a3d74a0926/master.m3u8?token=MGQ2MDA3NWM3NWJmMGQ
   4MQ==.HDv1J6sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ031Z_vYDWo1
   4fJ5czx10T1OkE0YMOPR7RwfFPRG4KG4qB_JOrdQ== "
}
Response - Failed
{
   "result": "failed",
   "message": "unauthorized"
}
```

Figure 12. Main playlist contents



Request

```
GET https://vod.wufat.com/94e99dc2-d5e2-4fb5-b294-
      f2a3d74a0926/master.m3u8?token=MGQ2MDA3NWM3NWJmMG
      Q4MQ==.HDvlJ6sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYD
      Wol4fJ5czx10TlOkE0YMOPR7RwfFPRG4KG4qB_JOrdQ==
```

Response - Success

```
#EXTM3U
# Created with Bento4 mp4-hls.py version 1.1.0r629

#EXT-X-VERSION:4

# Media Playlists
#EXT-X-STREAM-INF:AVERAGE-
BANDWIDTH=3292545,BANDWIDTH=5407632,CODECS="avc1.4D
4028,mp4a.40.2",FRAME-RATE=30.0,RESOLUTION=1920x1080
3d81d2ae-ee18-48c5-9167-
81ccc46f07d3/stream.m3u8?token=MGQ2MDA3NWM3NWJmMGQ4
MQ==.HDvlJ6sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5
czx10TlOkE0YMOPR7RwfFPRG4KG4qB_JOrdQ==
#EXT-X-STREAM-INF:AVERAGE-
BANDWIDTH=2674998,BANDWIDTH=4732837,CODECS="avc1.4D
401F,mp4a.40.2",FRAME-RATE=30.0,RESOLUTION=1280x720
488f02d1-256c-4d40-b42b-
354100bd4de8/stream.m3u8?token=MGQ2MDA3NWM3NWJmMGQ4
MQ==.HDvlJ6sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5
czx10TlOkE0YMOPR7RwfFPRG4KG4qB_JOrdQ==
#EXT-X-STREAM-INF:AVERAGE-
BANDWIDTH=1627704,BANDWIDTH=2675616,CODECS="avc1.4D
401E,mp4a.40.2",FRAME-RATE=30.0,RESOLUTION=720x480
df05075a-e6cb-4568-91d1-
85466e5db410/stream.m3u8?token=MGQ2MDA3NWM3NWJmMGQ4
MQ==.HDvlJ6sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5
czx10TlOkE0YMOPR7RwfFPRG4KG4qB_JOrdQ==
#EXT-X-STREAM-INF:AVERAGE-
BANDWIDTH=1003243,BANDWIDTH=1670442,CODECS="avc1.4D
401E,mp4a.40.2",FRAME-RATE=30.0,RESOLUTION=640x360
ef2cb965-e810-4b1b-8228-
062ed6047063/stream.m3u8?token=MGQ2MDA3NWM3NWJmMGQ4
MQ==.HDvlJ6sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5
czx10TlOkE0YMOPR7RwfFPRG4KG4qB_JOrdQ==
```

Response - Failed

```
HTTP 403
```

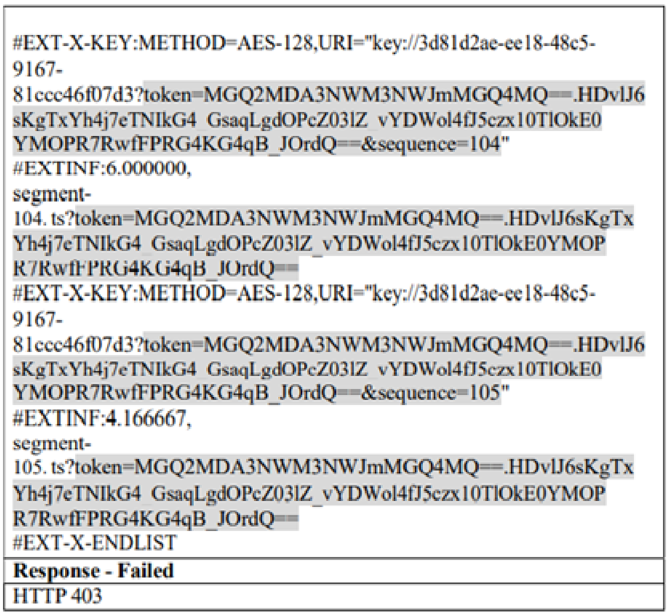
Request

```
GET https://vod.wufat.com/94e99dc2-d5e2-4fb5-b294-
      f2a3d74a0926/3d81d2ae-ee18-48c5-9167-
      81ccc46f07d3/stream.m3u8?token=MGQ2MDA3NWM3NWJmMG
      Q4MQ==.HDvlJ6sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYD
      Wol4fJ5czx10TlOkE0YMOPR7RwfFPRG4KG4qB_JOrdQ==
```

Response - Success

```
#EXTM3U
#EXT-X-VERSION:4
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-INDEPENDENT-SEGMENTS
#EXT-X-TARGETDURATION:6
#EXT-X-MEDIA-SEQUENCE:0
#EXT-X-KEY:METHOD=AES-128,URI="key://3d81d2ae-ee18-48c5-
9167-
81ccc46f07d3?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6
sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0
YMOPR7RwfFPRG4KG4qB_JOrdQ==&sequence=0"
#EXTINF:6.000000,
segment-
0.ts?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6sKgTxYh
4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0YMOPR7
RwfFPRG4KG4qB_JOrdQ==
#EXT-X-KEY:METHOD=AES-128,URI="key://3d81d2ae-ee18-48c5-
9167-
81ccc46f07d3?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6
sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0
YMOPR7RwfFPRG4KG4qB_JOrdQ==&sequence=1"
#EXTINF:6.000000,
segment-
1.ts?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6sKgTxYh
4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0YMOPR7
RwfFPRG4KG4qB_JOrdQ==

...
```

Figure 13. Clip playlist contents

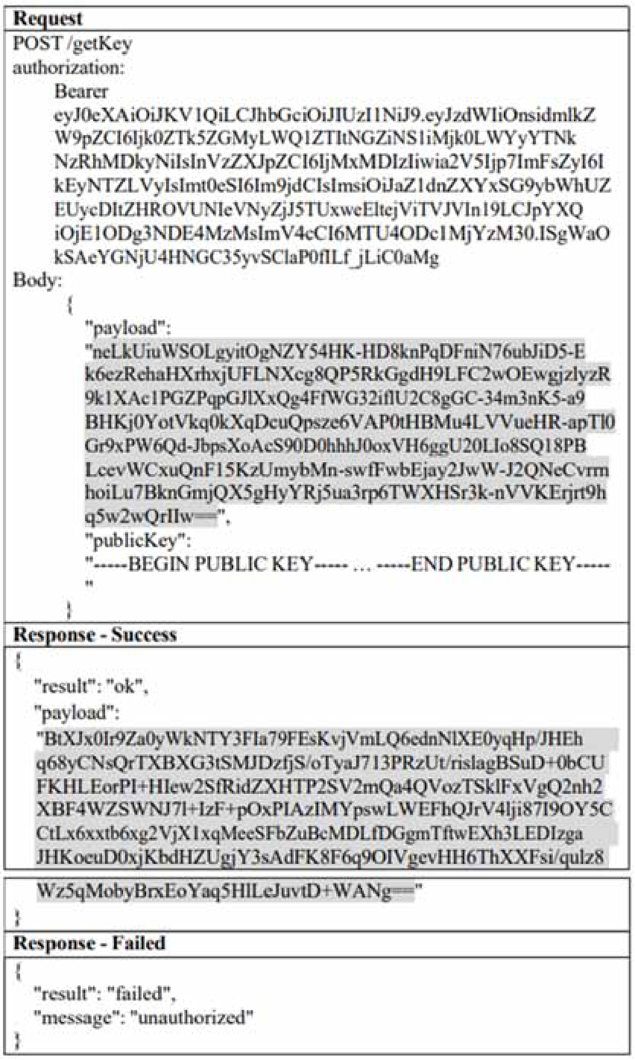


```
#EXT-X-KEY:METHOD=AES-128,URI="key://3d81d2ae-ee18-48c5-
9167-
81ccc46f07d3?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6
sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0
YMOPR7RwfFPRG4KG4qB_JOrdQ==&sequence=104"
#EXTINF:6.000000,
segment-
104. ts?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6sKgTx
Yh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0YMOP
R7RwfFPRG4KG4qB_JOrdQ==
#EXT-X-KEY:METHOD=AES-128,URI="key://3d81d2ae-ee18-48c5-
9167-
81ccc46f07d3?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6
sKgTxYh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0
YMOPR7RwfFPRG4KG4qB_JOrdQ==&sequence=105"
#EXTINF:4.166667,
segment-
105. ts?token=MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6sKgTx
Yh4j7eTNIkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0YMOP
R7RwfFPRG4KG4qB_JOrdQ==
#EXT-X-ENDLIST
```

Response - Failed

```
HTTP 403
```

Figure 14. Key request results



Request

```
POST /getKey
authorization:
      Bearer
      eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIiOnsidmlkZ
      W9pZCI6Ijk0ZTk5ZGMyLWQ1ZTItNGZiNS1iMjk0LWYyYTNk
      NzRhMDkyNiIsInVzZXJpZCI6IjMxMDIzIiwia2V5Ijp7ImFsZyI6I
      kEyNTZLVyIsImt0eSI6Im9jdCIsImsiOiJaZ1dnZXYxSG9ybWhUZ
      EUycDltZHROVUNIeVNyZjJ5TUxweEltejViTVJVIn19LCJpYXQ
      iOjE1ODg3NDE4MzMsImV4cCI6MTU4ODc1MjYzM30.ISgWaO
      kSAeYGNjU4HNGC35yvSClaP0flLf_jLiC0aMg
Body:
        {
          "payload":
          "neLkUiuWSOLgyitOgNZY54HK-HD8knPqDFniN76ubJiD5-E
          k6ezRehaHXrhxjUFLNXcg8QP5RkGgdH9LFC2wOEwgjzlyzR
          9k1XAc1PGZPqpGJlXxQg4FfWG32iflU2C8gGC-34m3nK5-a9
          BHKj0YotVkq0kXqDcuQpsze6VAP0tHBMu4LVVueHR-apTl0
          Gr9xPW6Qd-JbpsXoAcS90D0hhhJ0oxVH6ggU20LIo8SQ18PB
          LcevWCxuQnF15KzUmybMn-swfFwbEjay2JwW-J2QNeCvrm
          hoiLu7BknGmjQX5gHyYRj5ua3rp6TWXHSr3k-nVVKErjrt9h
          q5w2wQrIIw==",
          "publicKey":
          "-----BEGIN PUBLIC KEY----- … -----END PUBLIC KEY-----
          "
        }
```

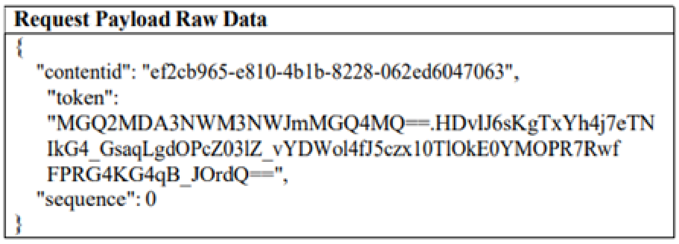
Response - Success

```
{
  "result": "ok",
  "payload":
  "BtXJx0Ir9Za0yWkNTY3FIa79FEsKvjVmLQ6ednNlXE0yqHp/JHEh
  q68yCNsQrTXBXG3tSMJDzfjS/oTyaJ713PRzUt/rislagBSuD+0bCU
  FKHLEorPI+HIew2SfRidZXHTP2SV2mQa4QVozTSklFxVgQ2nh2
  XBF4WZSWNJ7l+IzF+pOxPIAzIMYpswLWEFhQJrV4lji87I9OY5C
  CtLx6xxtb6xg2VjX1xqMeeSFbZuBcMDLfDGgmTftwEXh3LEDIzga
  JHKoeuD0xjKbdHZUgjY3sAdFK8F6q9OIVgevHH6ThXXFsi/qulz8
  Wz5qMobyBrxEoYaq5HlLeJuvtD+WANg=="
}
```

Response - Failed

```
{
  "result": "failed",
  "message": "unauthorized"
}
```

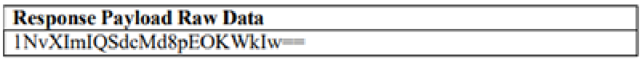
Figure 15. Key request raw data



Request Payload Raw Data

```
{
  "contentid": "ef2cb965-e810-4b1b-8228-062ed6047063",
    "token":
    "MGQ2MDA3NWM3NWJmMGQ4MQ==.HDvlJ6sKgTxYh4j7eTN
    IkG4_GsaqLgdOPcZ03lZ_vYDWol4fJ5czx10TlOkE0YMOPR7Rwf
    FPRG4KG4qB_JOrdQ==",
  "sequence": 0
}
```

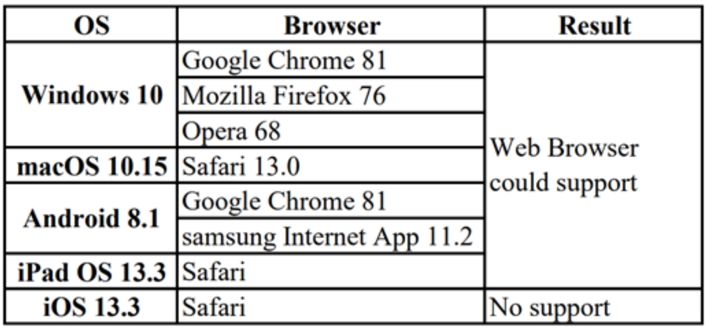
Figure 16. Key response raw data



Response Payload Raw Data

```
1NvXImIQSdcMd8pEOKWkIw==
```

Figure 17. Cross-platform test results



OS	Browser	Result
Windows 10	Google Chrome 81	Web Browser could support
	Mozilla Firefox 76	
	Opera 68	
macOS 10.15	Safari 13.0	
Android 8.1	Google Chrome 81	
	samsung Internet App 11.2	
iPad OS 13.3	Safari	
iOS 13.3	Safari	No support

Figure 18. Windows-Google Chrome playback

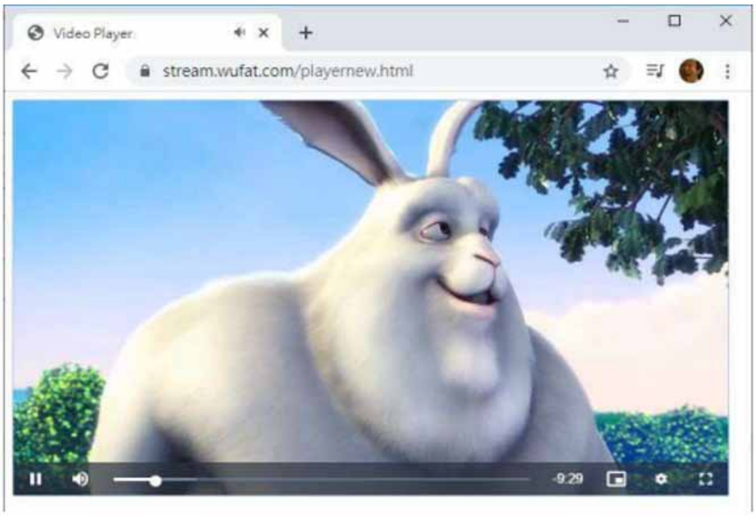


Figure 19. Windows-Mozilla Firefox playback

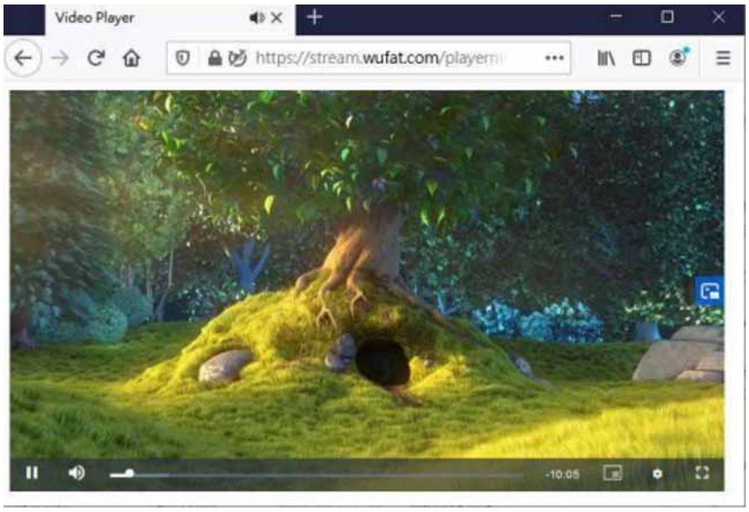


Figure 20. Android-Google Chrome playback

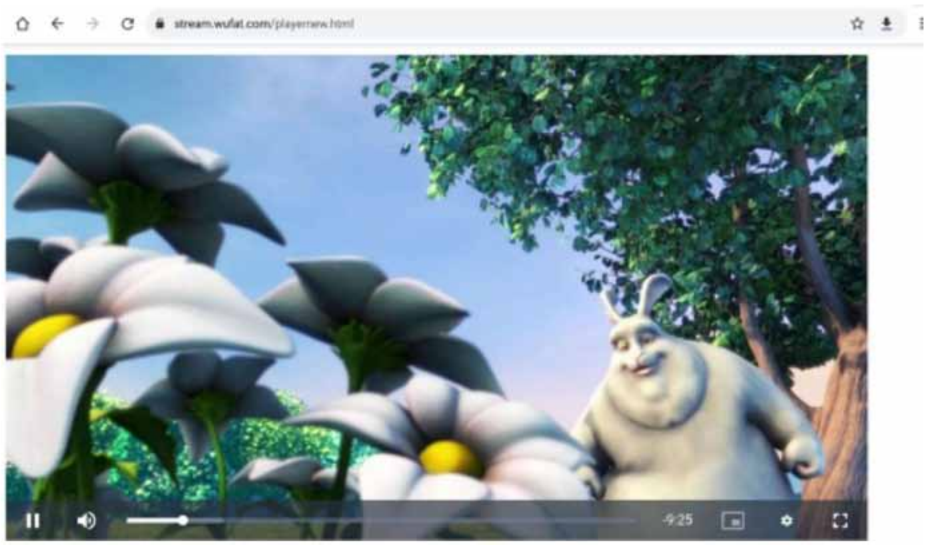
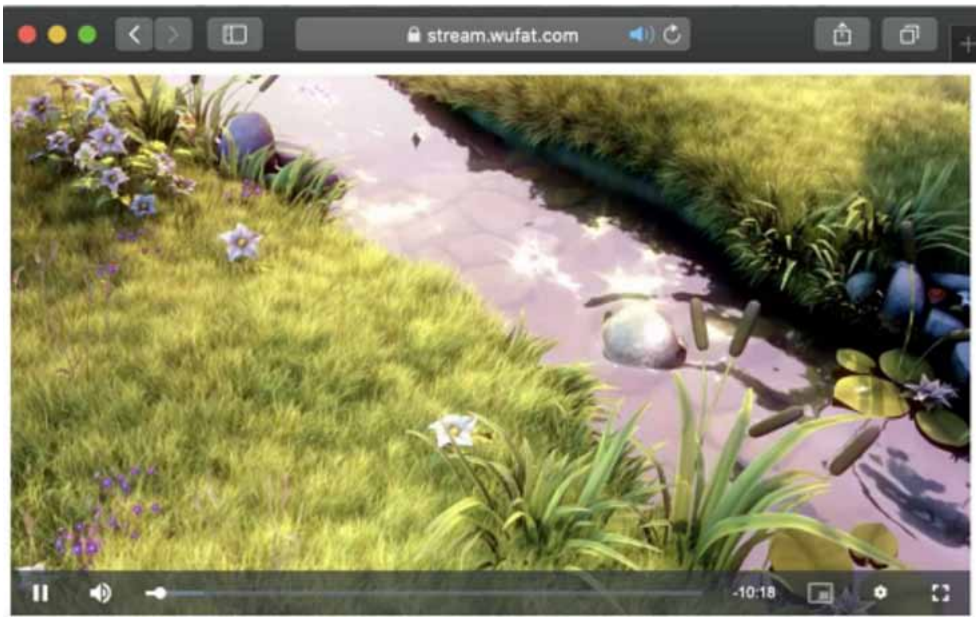


Figure 21. macOS Safari playback



it takes for the video server to perform dynamic AES-128 encryption for each video quality segment. The larger the file size is, the longer the encryption time.

b. Security Analysis: In this paper, our proposed methodology adopts the authentication server to authenticate users and allow successfully authenticated users to obtain the token. At the same time,

Figure 22. Android-Google Chrome playback

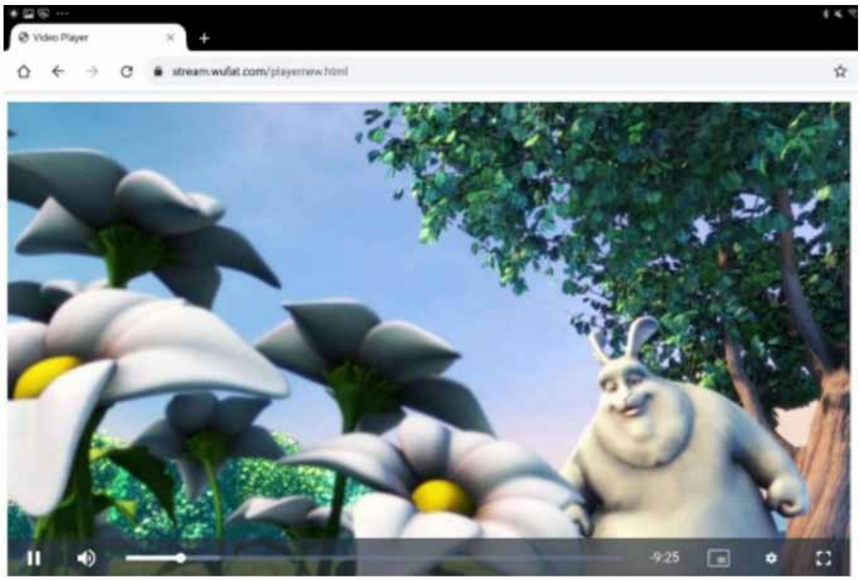
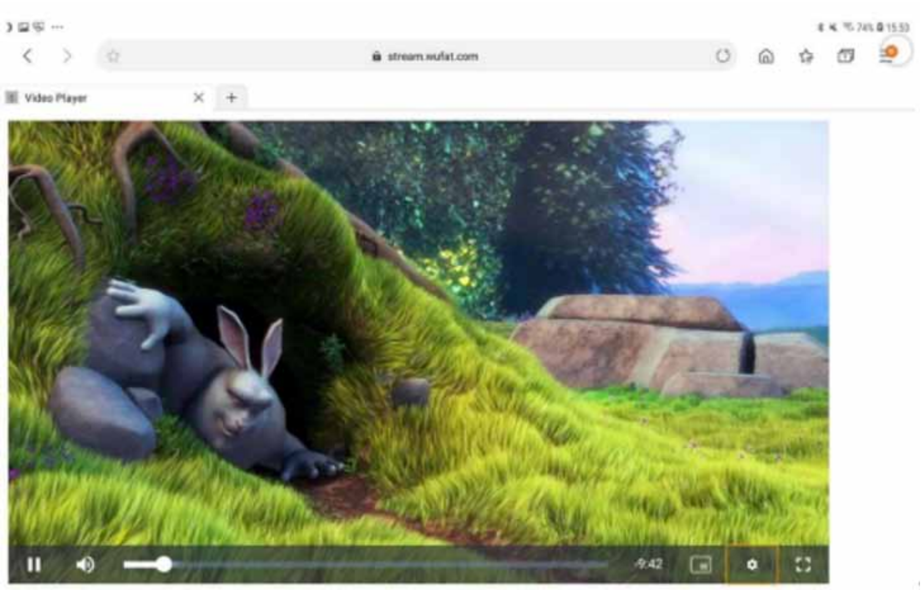


Figure 23. Android-Samsung internet playback



our method also takes RSA (Rivest et al., 1978) asymmetric encryption as the digital envelope to protect the AES-128 symmetric key. Adversaries cannot obtain the AES-128 decryption key by factoring the RSA encryption algorithm with nonnegligible probability. In addition, our method plays the video clip by fetching the AES-128 decryption key with authentication tokens, as demonstrated on different platform browsers. The user's browsers do not have to preserve the decryption key under different platforms and do not install any additional software to play the video clip.

Figure 24. Key server RSA encryption and decryption time

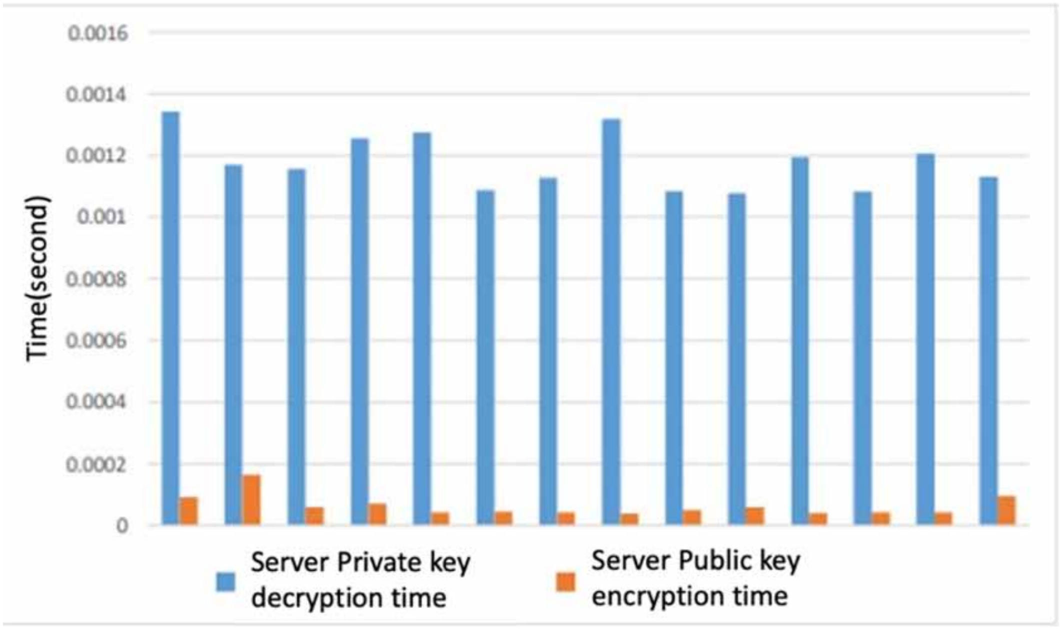
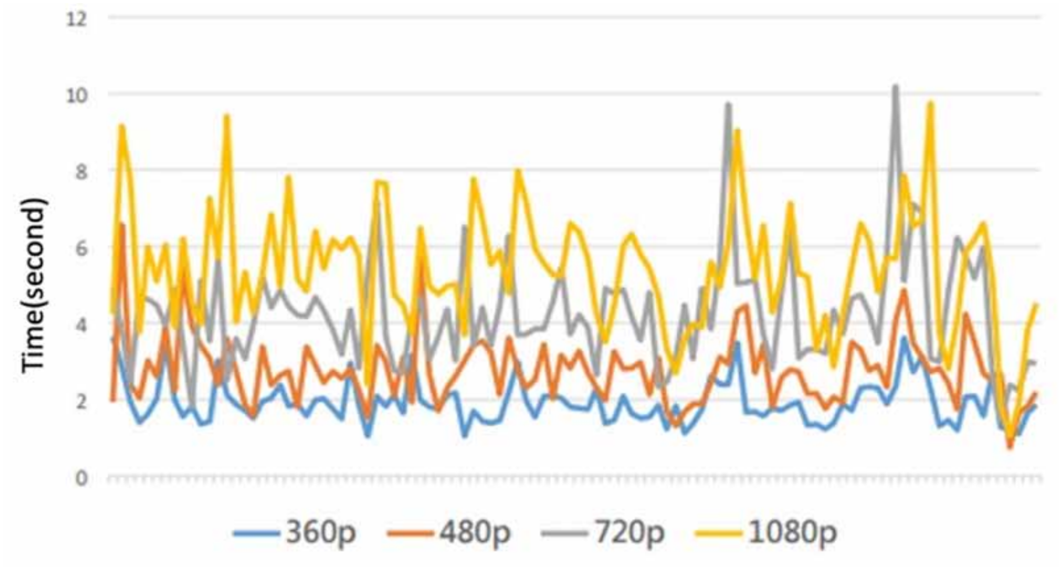
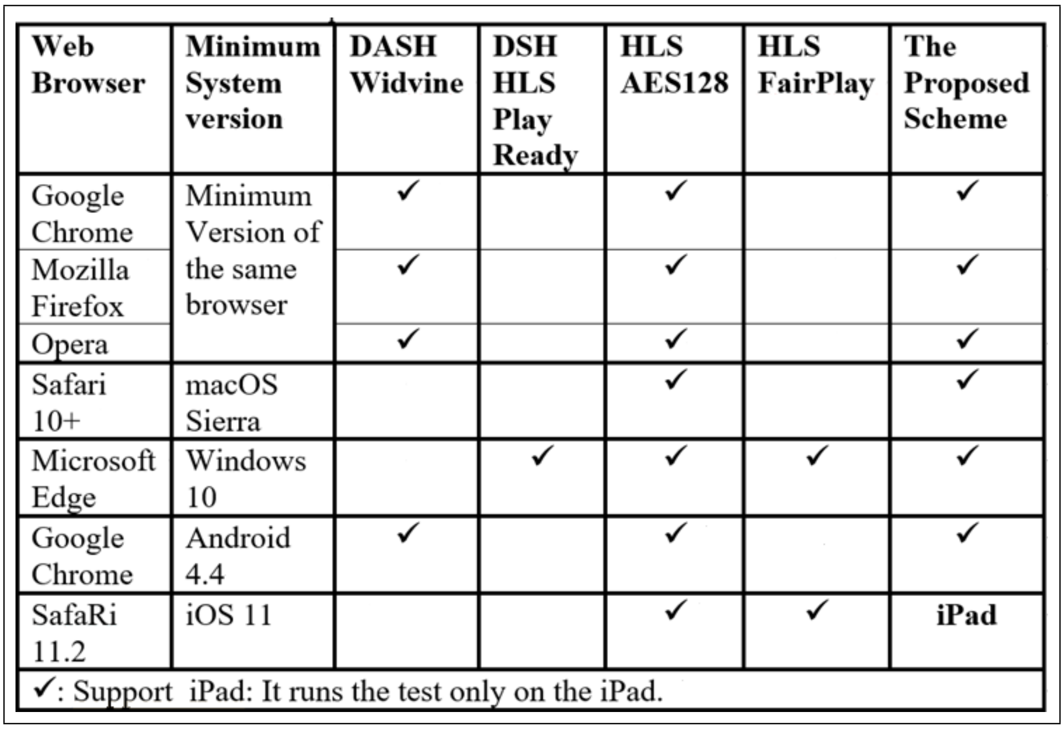


Figure 25. Video server dynamic encryption time spent



c. Comparison of methods: The proposed method supports browsers with various operating systems by executing WebAssembly and media source expansion in the browser. Table 1 compares the supported environment of this paper with other methods. The environment supports all browsers except for iPhone iOS, which does not support media source expansion, to a greater extent than the original Widevine and FairPlay supported environments.

Table 1. Comparison with other method support environments



Web Browser	Minimum System version	DASH Widvine	DSH HLS Play Ready	HLS AES128	HLS FairPlay	The Proposed Scheme
Google Chrome	Minimum Version of the same browser	✓		✓		✓
Mozilla Firefox		✓		✓		✓
Opera		✓		✓		✓
Safari 10+	macOS Sierra			✓		✓
Microsoft Edge	Windows 10		✓	✓	✓	✓
Google Chrome	Android 4.4	✓		✓		✓
SafaRi 11.2	iOS 11			✓	✓	**iPad**
✓: Support iPad: It runs the test only on the iPad.						

The method proposed by Hassan et al. (2020) is compared with the method proposed in this paper. The former uses a desktop application that the user must download before playing. Elliptic curve cryptography is used to encrypt the shared key and signature. In contrast, the proposed method is played directly through the browser and uses RSA encryption. Although the data generated by RSA is longer, research by Roetteler et al. (2017) indicates that the quantum computation required to crack ECC is less than that of RSA. The research by Hassan et al. Qubits can be cracked, but the RSA 2048-bit cracking used in this paper requires 4098 qubits, and n qubits can handle $2n$ information operations (Wikipedia, n.*d.).* Therefore, if quantum computers are successfully developed in the future, ECC will be cracked before RSA.

CONCLUSION AND FUTURE PROSPECTS

This paper proposes an adaptive streaming-based digital rights management system. This system authenticates the user's identity through the JSON Web Token, encrypts the request parameters with the server's RSA public key, and generates a set of RSA public and private keys from the user. When the server sends the request back, it is encrypted with the public key of the user side. The private key is available only on the user side, which it protects the request process from man-in-the-middle attacks and ensures that others cannot decrypt it after receiving it. The entire process is executed in WebAssembly to prevent direct access to the RSA private key and the video key. The video uses AES-128 dynamic encryption, and the key generated each time it is played is different, which makes it more difficult to crack. Only one format needs to be stored; it is no longer necessary to store multiple formats to support each platform device, thereby reducing the cost of storage space. If the browser supports WebAssembly and media source extensions, other browsers on Windows, macOS and Safari on iPad can be used. However, iOS Safari on iPhone, which does not yet support media source extensions, is not supported. Users do not need to install additional applications or settings, easing installation and use. Compared with direct transmission of AES-128 keys, the proposed method better protects the keys from direct access; it also supports more environments than Widevine or FairPlay do. At present, only software is used with the encryption protection key. If decryption can be combined with hardware to be independent of the operating system, it will be more difficult to crack. For example, the L1 of Widevine must be used with a Google-certified device. In the future, experiments regarding supporting low-latency live streaming and offline playback can be conducted.

ACKNOWLEDGMENT

Author Contributions: “Conceptualization, Ta-Jen Wu; Data curation, Ta-Jen Wu and Yi-Ying Chang; Formal analysis, Ming-Te Chen; Methodology, Yi-Ying Chang; Software, Ta-Jen Wu; Supervision, Yi-Ying Chang; Writing – original draft, Ming-Te Chen; Writing – review & editing, Ming-Te Chen.”

Conflicts of Interest: “The authors declare no conflict of interest.”

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

Apple Inc. (2016, Sept). *FairPlay Streaming*. https://developer.apple.com/streaming/fps/

Barker, E., & Barker, W. (2018). Recommendation for key management, part 2: best practices for key management organization (No. NIST Special Publication (SP) 800-57 Part 2 Rev. 1 (Draft)). National Institute of Standards and Technology.

Bitmovin Docs. (2019). *DRM Support*. https://developer.bitmovin.com/playback/docs/drm-content-protection

Dorwin, D., Smith, J., Watson, M., & Bateman, A. (2015). *Encrypted media extensions.* Academic Press.

Dworkin, M. (2016). Recommendation for block cipher modes of operation. *NIST Special Publication, 800*, 38G.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., & Bastien, J. F. et al. (2017, June). Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 185-200). doi:10.1145/3062341.3062363

Hassan, H. E. R., Tahoun, M., & ElTaweel, G. S. (2020). A robust computational DRM framework for protecting multimedia contents using AES and ECC. *Alexandria Engineering Journal*, *59*(3), 1275–1286. doi:10.1016/j.aej.2020.02.020

Hughes, K., & Singer, D. (2017). Information technology–Multimedia application format (MPEG-A)–Part 19: Common media application format (CMAF) for segmented media. *ISO/IEC, 19*, 23000.

Jian, S., Cao, D., Ximing, L., Ziyi, Z., Wenwen, W., Xiaoli, G., & Jin, Z. (2019). SELWasm: A Code Protection Mechanism for WebAssembly. *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 1099–1106. doi:10.1109/ISPA-BDCloud-SustainCom-SocialCom48970.2019.00157

Joan, D., & Vincent, R. (2002). *The design of Rijndael: AES-the advanced encryption standard.* Information Security and Cryptography.

Jones, M., Bradley, J., & Sakimura, N. (2015). *RFC 7519:* JSON Web Token (JWT).

Lederer, S. (2019) *Video Developer Report*. https://go.bitmovin.com/video-developer-report-2019

Martin, R., Michael, N., Krysta, S., & Kristin, L. (2017). Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms. *Advances in Cryptology - ASIACRYPT 2017*, 241–270. https://doi.org/.10.1007/978-3-319-70697-9_9

Nuttapong, A., Goichiro, H., Shigeo, M., Yusuke, S., Kana, S., & Tadanori, T. (2018). Efficient Two-Level Homomorphic Encryption in Prime-Order Bilinear Groups and A Fast Implementation in WebAssembly. *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018,* 685–697. https://doi.org/ doi:10.1145/3196494.3196552

Overdigital. (2016). *Streaming Format Evolution — Does CMAF Give Us the Single Format?* https://www.overdigital.net/2016/06/18/cmaf-streaming-format-evolution/

Pantos, R., & May, W. (2017). *RFC 8216: HTTP live streaming*. Academic Press.

Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, *21*(2), 120–126. doi:10.1145/359340.359342

Rossberg, A. (2019, Oct). *WebAssembly core specification.* W3C First Public Working Draft.

Schmidt, O. (2020). WebCloud: Web-Based Cloud Storage for Secure Data Sharing Across Platforms. *IEEE Transactions on Dependable and Secure Computing*, *19*(3), 1871–1884. doi:10.1109/TDSC.2020.3040784

Stockhammer, T. (2011, February). Dynamic adaptive streaming over HTTP— standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems* (pp. 133-144). doi:10.1145/1943552.1943572

SYNOPI. (2016). *What Is HLS (HTTP Live Streaming)? How HLS Works?* https://www.synopi.com/hls-http-live-streaming/

Taiwan Network Information Center. (2019). *Taiwan Internet Report*. https://report.twnic.tw/2019/assets/download/TWNIC_TaiwanInternetReport_2019_EN.pdf

Wikipedia contributors. (2023). Qubit. In *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Qubit&oldid=1186529789

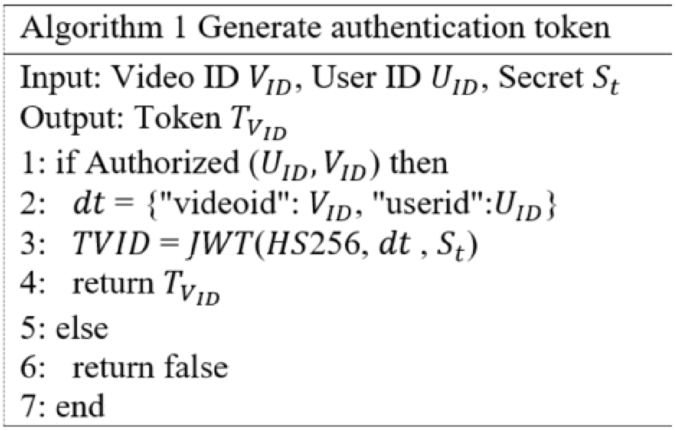
Wu, T. J. (2020). *A Cross Platform Digital Rights Management Based on Dynamic Encryption* [Unpublished master's thesis]. National Chin-Yi University of Technology, Taichung, Taiwan.

Zishuai, S., Hui, M., Rui, Z., Wenhan, X., & Jianhao, L. (2023). Everything Under Control: Secure Data Sharing Mechanism for Cloud-Edge Computing. *IEEE Transactions on Information Forensics and Security*, *18*, 2234–2249. doi:10.1109/TIFS.2023.3266164

APPENDIX

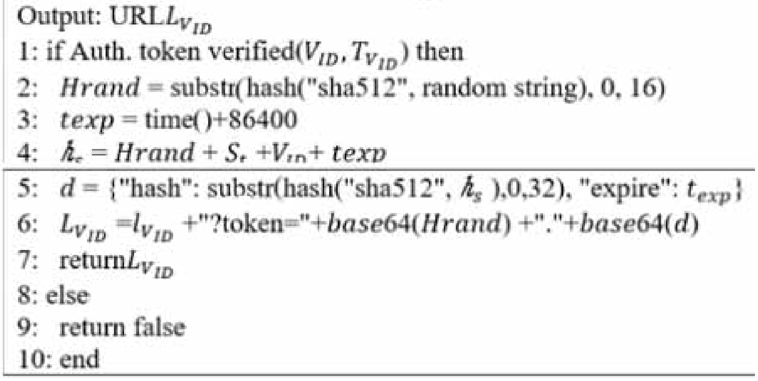
1. Experimental Method:
a. Certified Token Generation: The party that generates the authentication token returns a set of JSON web token by confirming that the user can play the video. This JWT data verification uses HS256, and the data field contains the video ID and user ID. Other APIs can use this authentication token to identify licensed videos and objects in Figure 26.
b. Playback URL Generation: When requesting the video playback URL, the token generated in 3.2.1 is used to make the request. After the token is verified, the corresponding video URL will be generated. The URL contains the URL Token, which contains a set of random numbers and encrypted data fields. The field is encrypted with AES-256-ECB, and the field contains the URL expiration time and the SHA512 hash value calculated by random numbers, video ID, expiration time and secret string to confirm that the token is legitimate in Figure 27.
c. Playlist generation: In each playlist, there is a URL for each clip of the quality, and the URL token and clip number are added after the URL in Figure 28 and Figure 29.

Figure 26. Certified token generation



```
Algorithm 1 Generate authentication token
Input: Video ID $V_{ID}$, User ID $U_{ID}$, Secret $S_t$
Output: Token $T_{V_{ID}}$
1: if Authorized ($U_{ID}, V_{ID}$) then
2:    $dt$ = {"videoid": $V_{ID}$, "userid":$U_{ID}$}
3:    $TVID = JWT(HS256, dt , S_t)$
4:    return $T_{V_{ID}}$
5: else
6:    return false
7: end
```

Figure 27. Playback URL generation



```
Algorithm 2 Generate playback URL
Input: Video ID $V_{ID}$, Auth. token $T_{V_{ID}}$, Secret $S_t$
Output: URL$L_{V_{ID}}$
1: if Auth. token verified($V_{ID}, T_{V_{ID}}$) then
2:    $Hrand$ = substr(hash("sha512", random string), 0, 16)
3:    $texp$ = time()+86400
4:    $\hbar_s = Hrand + S_t + V_{ID} + texp$
5:    $d$ = {"hash": substr(hash("sha512", $\hbar_s$ ),0,32), "expire": $t_{exp}$}
6:    $L_{V_{ID}}$ =$l_{V_{ID}}$ +"?token="+$base64(Hrand)$ +"."+$base64(d)$
7:    return$L_{V_{ID}}$
8: else
9:    return false
10: end
```

d. Key Generation:
e. The fragment output generates the fragment encryption key according to the random hash code in the provided URL token in Figure 30.
f. Key Delivery:

Figure 28. Master playlist generation

```
Algorithm 3 Generate master playlist
Input: Video ID $V_{ID}$, URL token $T_u$, Master playlist $P_m$
Output: Modified master playlist $P_m$
1: if URL Token verified($V_{ID}$ , $T_u$) then
2:    for Segment playlist URL in $P_m$
3:       Segment playlist URL+="?token="+$T_u$
4:    end
5:    return $P_m$
6: else
7:    return false
8: end
```

Figure 29. Segment playlist generation

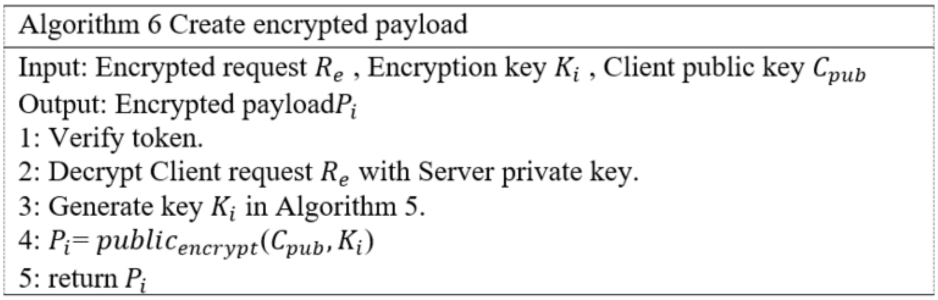
```
Algorithm 4 Generate segment playlist
Input: Video ID $V_{ID}$, URL token $T_u$, Segment playlist $P_s$
Output: Modified segment playlist $P_s$
1: if URL Token verified ($V_{ID}$ , $T_u$) then
2:    for Segment URL, Segment key URL, Sequence index in $P_s$
3:       Segment URL+="?token="+$T_u$
4:       Segment key URL+="?token="+$T_u$+"&sequence="+Index
5:    end
6:    return $P_s$
7: else
8:    return false
9: end
```

Figure 30. Key generation

```
Algorithm 5 Generate segment encryption key
Input: Video ID $V_{ID}$, URL token $T_u$, Segment index $i_{index}$ , Secret $S_t$
Output: Segment key $K_i$
1: Verify $T_u$
2: Extract $k_{salt}$ from $T_u$
3: $h_{key} = k_{salt} + S_t + i_{index}$
4:$K_i$ = substr(hash("sha512", $h_{key}$),0,32)
5: return$K_i$
```

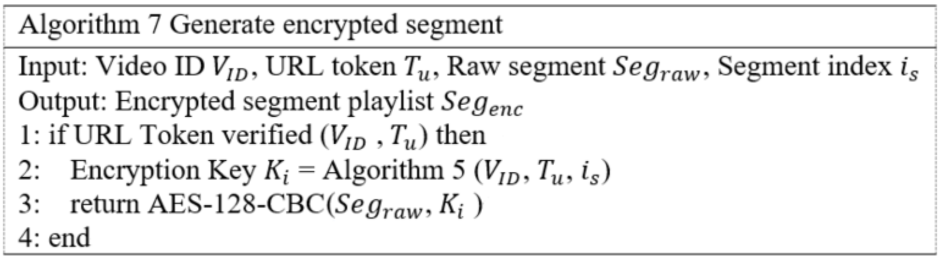
The keys delivery shows in the Figure 31.
g. Encrypted Fragment Generation:
h. The encrypted segment is requested through the segment URL in the segment playlist obtained in 3.3.3. The server verifies the URL token, generates a key according to 3.3.4, and then encrypts the key using AES-128-CBC according to the HLS standard definition in Figure 32.

Figure 31. Key delivery



```
Algorithm 6 Create encrypted payload
Input: Encrypted request $R_e$ , Encryption key $K_i$ , Client public key $C_{pub}$
Output: Encrypted payload$P_i$
1: Verify token.
2: Decrypt Client request $R_e$ with Server private key.
3: Generate key $K_i$ in Algorithm 5.
4: $P_i = public_{encrypt}(C_{pub}, K_i)$
5: return $P_i$
```

Figure 32. Encrypted segment



```
Algorithm 7 Generate encrypted segment
Input: Video ID $V_{ID}$, URL token $T_u$, Raw segment $Seg_{raw}$, Segment index $i_s$
Output: Encrypted segment playlist $Seg_{enc}$
1: if URL Token verified ($V_{ID}$ , $T_u$) then
2:    Encryption Key $K_i$ = Algorithm 5 ($V_{ID}$, $T_u$, $i_s$)
3:    return AES-128-CBC($Seg_{raw}$, $K_i$ )
4: end
```

Ming-Te Chen was born in Tainan, Taiwan on August 2, 1980. He received his M.S. degree in computer science and information engineering from National Sun Yat-sen University, Taiwan, in 2005, and a Ph.D. degree in computer science and information engineering from National Sun Yat-sen University in 2012. In 2018, he joined the faculty of the Department of Computer Science and Information Engineering, National Chin-Yi University Technology, Taichung, Taiwan. His current research interests include information security, applied cryptographic protocols, digital signatures, IoT security, and electronic commerce.