DASH Video Authentication- Final Report

Daniel Kester
dkester3@gatech.edu

1. Introduction

In recent years, video streaming has become the dominant source of Internet traffic, with Netflix alone as the source of 35% of the United States' total Internet traffic in 2014 [5]. The most popular modern method of streaming videos is known as Dynamic Adaptive Streaming over HTTP, or DASH. In DASH, videos are divided into segments that are downloaded by video players using HTTP rather than a custom protocol [4]. This has many advantages, including the fact that security can be done using TLS in a web server rather than building it into the player. However, when all a user wants is authentication, TLS can incur excessive overhead as it works at the packet layer and also encrypts the data. By taking advantage of the DASH manifest (MPD) file, we may be able to more efficiently authenticate the video being streamed.

2. Objectives

The main goal of this project is to create an authenticated DASH video player. This will be accomplished through three main steps. The first step is to create a program that could update a DASH MPD file to include authentication information needed by the client. The second step is to modify an open source DASH player to use the updated MPD to authenticate the video as it is being played. The third and final step is to measure the updated player's performance and compare it to that of Transport Layer Security (TLS), the default authentication used by DASH.

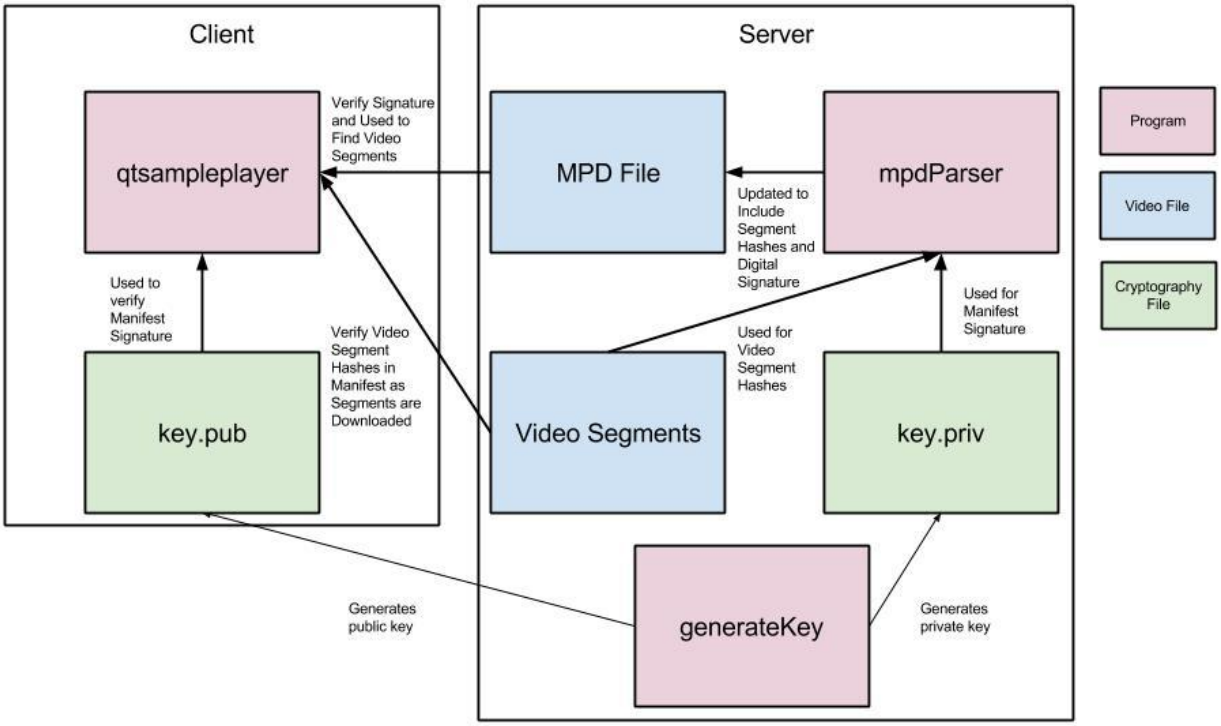
3. Algorithm

There are two algorithms used in my project. The first algorithm is used to update the MPD to include authentication information for the video. It first goes through each video segment listed in the MPD, calculates the SHA-512 hash of that file, and adds the hash as an attribute in the MPD. Once every segment's hash has been added, it finally takes an RSA digital signature of the modified MPD using the server's private key and adds it in as a tag at the end of the MPD.

The second algorithm is used to authenticate the video by the player and is largely the reverse of the modification algorithm. First, it uses the server's public key to verify the RSA digital signature included at the end of the MPD. If the signature matches the file and the key, the MPD is trusted and the video can begin playback. Otherwise, the MPD is not trusted- the player throws an error and closes. Once playback begins using the trusted MPD, the SHA-512 hash is calculated for each segment as they are downloaded. If this hash matches the hash listed in the trusted MPD, playback continues. Otherwise, the player throws an error and closes.

The security of this algorithm comes in two parts. First, by verifying the digital signature in the MPD using the server's public key, the client can insure that the MPD did indeed come from the server. Once the MPD is trusted, we can trust the hash values listed in the MPD. If the hash of a video segment matches the hash in the MPD, we can trust it has come from the same server that we got the trusted MPD from. By combining these two steps, we can ensure that the entire video comes from the server of the public key.

4. Architecture



The authenticated DASH player uses the standard client/server architecture like that of any typical DASH player. On the server side, there is the GenerateKey program (which creates the cryptography keys) and MpdParser (which updates the DASH MPD to include authentication information). On the client side, QtSamplePlayer is the modified open source DASH player that will authenticate a video based on the updated MPD. These three programs are discussed in detail below.

4.1. GenerateKey

GenerateKey is the most straightforward program written for my authenticated DASH player. All it does is generate a public and private RSA key pair using the Crypto++ library. These keys are saved as <name>.pub and <name>.priv, where <name> is a string passed into the program.

4.2. MpdParser

MpdParser is the server program that updates a MPD to include authentication information. This involves two main steps- adding hashes for the video segments and adding the digital signature.

The first part of the MpdParser adds a hash for each video segment. Using the C++ header library RapidXML, the MpdParser recursively visits each node within the XML DASH MPD file. Each time the parser finds an Initialization or SegmentURL tag, it must create a new hash. To do this, it combines the base directory passed into MpdParser as an argument with the path given in the XML tag to find the file to hash. It then takes that file and creates a SHA-512 hash of it using the Crypto++ library. Finally, it adds the calculated hash as an attribute to that tag in the MPD.

The second part of the MpdParser creates the digital signature of the file. After all the hashes have been added as described above, the MpdParser writes the updated MPD as a temporary file. It then reads the modified file in as the source of the digital signature. Using the private key passed in as an argument, the MpdParser then creates an RSA digital signature of the modified file. This signature is then put inside of a Signature XML tag and added at the end of the MPD. At this point, the MPD has been updated to include everything needed for the video player to authenticate the video.

The following is a sample MPD file that has been modified by MpdParser. The red bolded text is what has been added.

```
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500000S" type="static"
mediaPresentationDuration="PT0H3M1.44S" profiles="urn:mpeg:dash:profile:full:2011">
    <ProgramInformation moreInformationURL="http://gpac.sourceforge.net">
        <Title>carFull_dash.mpd generated by GPAC</Title>
    </ProgramInformation>
    <Period id="" duration="PT0H3M1.44S">
        <AdaptationSet segmentAlignment="true" maxWidth="426" maxHeight="240" maxFrameRate="90000/3754"
par="426:240">
            <Representation id="1" mimeType="video/mp4" codecs="avc1.4d4015" width="426" height="240"
frameRate="90000/3754" sar="1:1" startWithSAP="1" bandwidth="265102">
                <SegmentList timescale="90000" duration="0">
                    <Initialization sourceURL="carinit.mp4"
hash="4D4AFB6D58DF0CA42AA9E2BD9696555FD6C5E828CE27BC845BC383D9F5A0F17FF5E096F077B618B23FB76E5E22
0FCCA43600EFAADAB237FEE9940DC51421BCB6"/>
                    <SegmentURL media="car1.m4s"
hash="268EF5276F1DE52CF4EA0922203C998CD050E3CE614095F33AC5541F65D59450B27C34EAAB54D9D658A874E5692
B5B6BDBD3A075550580B72FAD34164039D87A"/>
                </SegmentList>
            </Representation>
        </AdaptationSet>
    </Period>
<Signature>C35F781D0A744A0D82DFADA35ECE2DAB7944E5EC5AD007E9AC649AF49F253A05087D719C09FAC4DB0F62E
FB3F7CA563107F6A8AFDC80C81B653BB768103247BD8AA4C3498BF1C01175A1451D264B66F23F4A93BF1A7275F2B3C0E
FEC980A328CD0070C9120CAD428B0BD1A0DF3384088A04A8E341F71E1840D81E8A680C8578C</Signature>
</MPD>
```

4.3. QtSamplePlayer

QtSamplePlayer is the open source DASH video player I modified to do video authentication as it is playing. QtSamplePlayer started as a demo video player for the libdash C++ library and was originally designed around demoing the features of the library. There were three main changes I made to make QtSamplePlayer authenticate videos- remembering the hash and signature information from the MPD, verifying the digital signature of the MPD, and validating the hashes of the video segments as the segments arrive.

The first change I made to QtSamplePlayer was to ensure that the player correctly read in and remembered the digital signature and segment hashes listed in the MPD. By default, the libdash library reads in the MPD, saves all the information needed for a DASH video player and throws away the rest. In order to save the hash and signature information, I had to alter libdash itself to keep track of this information. In the file Node.cpp, I added a section to the ToURLType method that sets the URLType's hash value if it exists. I also added a section to the method ToMPD that adds the signature to the MPD class if the signature tag exists in the MPD. These two changes allowed me to keep track of the extra authentication information going forward into the open source project.

The next change I made was to have the player authenticate the MPD's digital signature when the MPD is downloaded. This authentication was done by modifying the file MultimediaManager.cpp. In the method Init, before having libdash process a copy of the MPD, I had the file create a copy of the MPD without the digital signature. Using this copy of the file with the signature removed, the passed in public key and the RSA digital signature that libdash had read from the file, the method validates the digital signature using Crypto++. If the signature is valid, QtSamplePlayer trusts that the MPD came from the

server and continues as it normally does. If the signature is invalid (either because the MPD has been modified or was not signed with the matching private key), the player is closed and the console throws an error.

The third and final change I made was to have the player validate the hashes of the video segments as they are played. This validation is done in the file MediaObject.cpp. In this file, the method Read reads the segment directly off the web server like a standard sockets read. Since I needed the entire file to do the hash validation and the Read method only grabs a small chunk of the file at once, I saved a copy of that chunk to a vector called segmentBytes. Once we get to the end of the segment (as indicated by the server returning 0 bytes read), the segmentBytes vector corresponds to the entire video segment. At this point, we can feed segmentBytes into Crypto++ and calculate the SHA-512 hash of the downloaded segment. If this hash matches the hash given in the trusted MPD file, the segment is considered valid and the player continues as normal. If the hash does not match the hash given in the trusted MPD file, the segment is considered invalid, the player is closed, and an error is thrown in the console.

5. Challenges

During the course of this project, there were five challenges I encountered- debugging cryptography, using a consistent key format, reading through unfamiliar open source code, learning new technologies, and dealing with a problem with QtSamplePlayer's buffer. Each of these challenges are described in detail below.

One major challenge I encountered in this project was debugging cryptography. Cryptography, as a concept, is designed to be as confusing as possible to make it more difficult for an adversary to break. Because of this confusion, debugging cryptography in my project was a major challenge. The only real feedback returned by the Crypto++ library was that the digital signature was a success or it was a failure, without any real kind of message indicating the cause of the problem. In the end I was finally able to get the digital signature and hash validations to work correctly, but it took much more time to debug than problems I've encountered in other fields (such as web development).

Another major challenge I encountered was using consistent key formats for the digital signature. When discussing cryptography from a theoretical point of view, an RSA key is the same no matter where it comes from, as it just represents a large number used in an equation. Because of this theory, I originally planned on using simple Linux SSL key for my digital signature. However, in practice, different cryptography libraries expect these keys to be in a specific format. When I tried using the Linux keys with Crypto++, the library couldn't understand the format of these SSL keys. In the end, I had to write the GenerateKey program, which creates a public and private key pair specifically for use with Crypto++, even if they were functionally equivalent to the Linux keys.

A third challenge I encountered was working with an unfamiliar open source project. QtSamplePlayer and libdash together have over 100 files of code divided up in many different folders, much more than I have encountered anywhere other than my place of work. Because of the large number of unfamiliar files, I had to spend a lot of my time simply finding the correct method that my modifications needed to be placed within. This discovery involved starting from QtSamplePlayer's main.cpp and stepping through method calls across files until I finally found where the MPD and segments were downloaded. This process probably took up most of the time I spent working on QtSamplePlayer, as the actual authentication modifications were fairly straightforward to write. While this process was challenging, it was good experience for future work I'm likely to run into at my job.

The fourth challenge was learning new technologies involved with QtSamplePlayer. For example, QtSamplePlayer is compiled using cmake. Cmake is a program that uses some options provided

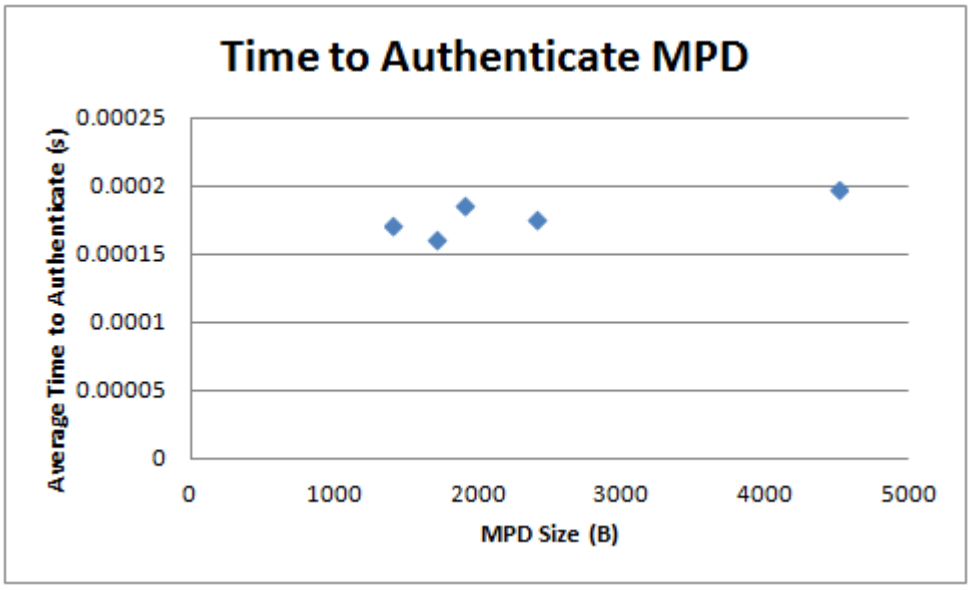
by the user to create a makefile for compiling the project. Since cmake does not know the location of the Crypto++ library and does not have the C++11 flags set by default, I had to learn enough about the system in order to make sure these two modifications were added to the generated makefile. While learning this process took some time, in the end it was worth it.

The final and most significant challenge I encountered while working on this project was QtSamplePlayer's incorrect video buffering. In a typical video buffer as described in a networking class, video segments are downloaded ahead of time by the player and stored until they are needed for playback. This way, even if there are delays on the network, the video itself appears uninterrupted by the user. However, the "video buffer" used by QtSamplePlayer instead stores URLs of video segments. When a segment is needed to play, QtSamplePlayer just does the equivalence of a sockets read to get the segment directly off of the server. This caused two major problems with project. First, this incorrect buffer means I was unable to do the hash validation until after the video segment has been played. Because the player does not actually have the entire segment until it is done being played, I had to wait for the segment to finish in the player before I could check the segment's hash. This means that in addition to being unable to view future video segments when there is an authentication problem, the user must now also remember to not trust the segments they have already seen. This problem is an inconvenience, but it does not prevent the player from correctly identifying authentication issues. The second issue I encountered was that without a buffer, the cryptography had to be done between video segments rather than behind the scenes. This meant that if the cryptography delay was long enough, there would be an unacceptable pause between video segments. While this was a major potential problem, it turned out that typical video segment sizes were short enough that the delay caused by the cryptography was not visible to the human eye. In addition, the ratio of the cryptography delay to the segment video length was small enough that there likely would not have been a problem if a buffer existed. For more details, see section 6.4 below.

6. Performance

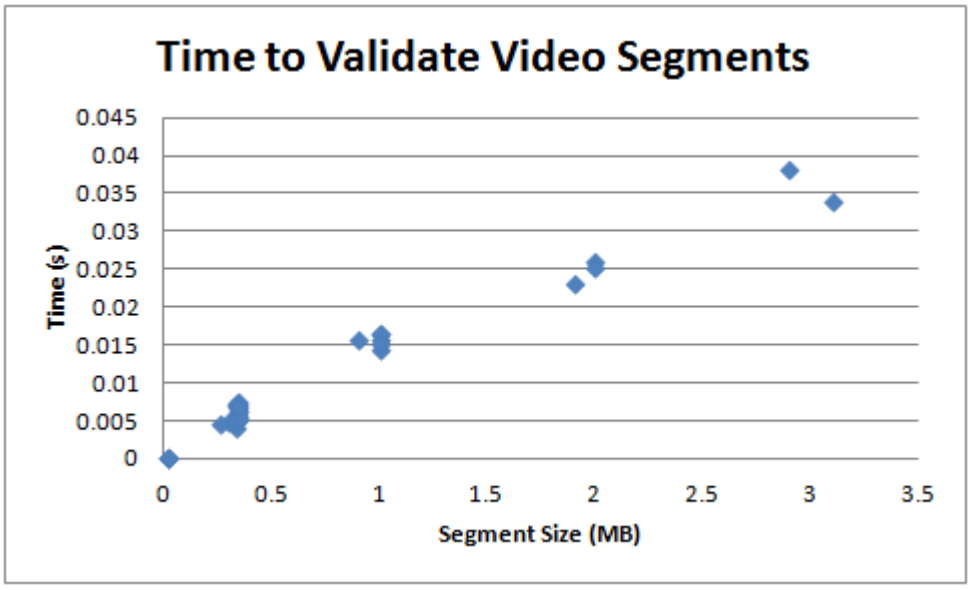
To measure the performance of my DASH player, I used the timing tools in C++11's chrono.h. This header includes ways to get the current time down to the nanosecond using its now method. I subtracted the difference between the time before I did the cryptography calculations and after the calculations to see how much extra overhead my system provided. I recorded this difference 5 times for each video segment and averaged the times together for the graphs below. The video itself was 181 seconds long, and was divided into segments of 10, 30, 60, 90, and 181 seconds using MP4Box for the different experiments. For the full timing results, see the file CS 6235 Performance.xlsx.

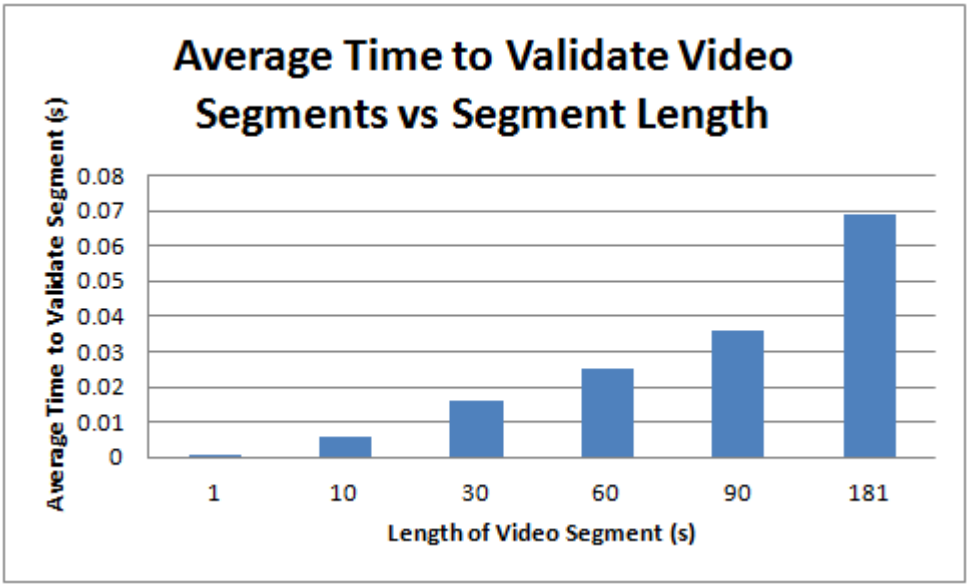
6.1. MPD Authentication



This graph shows the average amount of time it took to authenticate the digital signature of each MPD file that I tested. The main takeaway of this graph is that the time it took to authenticate the signature was extremely small (less than a millisecond), even smaller than the time to do the hash validations. This is contrary to what I originally believed would happen, as the public key cryptography of RSA usually takes much longer than the secret key cryptography of SHA-512. What I hadn't considered, however, is how much smaller the MPD file sizes are than the video segments (on the order of kilobytes as opposed to megabytes). Since the performance of cryptography is directly related to the size of the file, this size difference explains why the signature authentication was so much faster than the hash validations.

6.2. Hash Validation





These two graphs look at the performance of the hash validation process. The first graph shows the average time to validate a segment against the file size of that segment. The second graph averages the segments of the same video length together (in other words, the second graph is an average of the clusters on the first graph). There are two main takeaways from these graphs. First, the growth of the delay is fairly linear, so the proportion of the time to validate the segments to the length of the segments is fairly consistent. The second takeaway is that the actual time to validate the segments is extremely small compared to the length of the video segment itself. For a 1 second segment it is about .5 milliseconds, for 10 seconds it is about 5 milliseconds and for a minute of video it is about 25 milliseconds. This is in the order of a few thousandths of the original video segment, which is several orders of magnitude smaller. This is extremely good, as short cryptography delays help ensure the video continues to play smoothly.

6.3. Comparison to TLS

In order to see just how well my system works, I will compare it to Transport Layer Security (TLS), the default authentication and encryption used by HTTPS. While I couldn't find any good delay studies of TLS, I did find a study of its byte overhead. This study found that on average, TLS adds about 300 bytes to a TCP packet [1]. Using this information, I can calculate the total overhead added for a video using the following equation:

TLS Overhead = File Size / Bytes per TCP Packet * Average TLS Overhead per TCP Packet

I can then calculate the overhead of my system by looking at the bytes added into the MPD file.

DASH Authentication Overhead = Number of Segments * 136 Hash Validation Bytes per Segment + 281 Signature Bytes per Video

Using these two equations, we can compare the two systems. First, I will compare the best case of TLS to the worst case of my system for a one minute video (about 2 MB of video). For TLS, the best case is when it is using full 64 kilobyte TCP packets. This gives you the following calculation:

2 MB of video / 64 KB per packet * 300 bytes of TLS per packet = **9600 bytes**

For my system, the worst case is a video divided into 1 second segments (which is small enough that the open source player starts to run into performance issues). For my DASH Authentication system, the worst case overhead for a one minute video is:

60 video segments * 136 Hash Validation Bytes per Segment + 281 Signature Bytes per Video = **8441 bytes**

The worst case of my system's 8441 bytes of overhead is still smaller than the 9600 bytes of overhead for TLS, but not significantly. However, when you consider the fact that this is the best case versus the worst case, the fact that my system is still better is extremely good.

Next, I will compare the overhead of the two systems in a more typical case. On average TCP packets tend to be much smaller than the full 64 KB packets, closer to about 576 bytes per packet [2] gives you the following equation:

2 MB of video / 576 B per packet * 300 bytes of TLS per packet = **1.04 megabytes**

A more typical segment length for a DASH video is about 4 seconds [3]. Using this number, we can do the following calculation:

15 video segments * 136 Hash Validation Bytes per Segment + 281 Signature Bytes per Video = **2321 bytes**

So in the typical case, my system's 2321 bytes of overhead is significantly smaller than the 1.04 megabytes of overhead of TLS.

The reason my system ends up working so much better than TLS in general is twofold. First, TLS does both encryption and authentication, so there is extra overhead that is not needed by my system that only does authentication. Second, TLS operates at the packet level, whereas my system operates at the video segment level. This means that the TLS overhead gets added much more often than on my system, especially since TCP packets tend to be smaller and not as efficiently used as in the best case, leading to an excessive amount of overhead. These two reasons combined explain why my system works so well.

6.4. Video Buffer

While QtSamplePlayer did not buffer correctly so I cannot definitely show that I would be able to hide the cryptography delay behind the buffer, I can argue that this would be the case using my performance measurements. First of all, typical DASH segments tend to be smaller [3], on the order of a few seconds than a few minutes. From my experiments, the delay between segments isn't noticeable until you start hitting segments that are about 2.5 minutes long, so typical small DASH segments would not have to worry about the delay regardless of the buffer status. For longer segments, we simply have to look at the ratio of time to do a hash validation to the video segment itself. For example, look at the YouTube buffer which is about 1 minute long (this can be tested by pausing a video and seeing how far ahead YouTube buffers). From my measurements, we know that one minute of video takes about 25 milliseconds to validate, or about .04% of the length of the video. If a buffer cannot handle a variation of .04%, then the buffer simply will not work. This is because .04% is small enough that it could easily be

caused by a network delay. If a buffer cannot handle typical network delay, it will be unable to accomplish its goal of hiding those delays and jitter will show up in the video. Because a buffer must be able to handle such small fluctuations, it stands to reason it would be able to handle the small delay caused by cryptography. These two reasons combined are why I believe that a buffered version of my system would be able to easily hide the cryptography delay behind the buffer.

7. Most Interesting Contributions Learned from Hands On Experience

There are two major interesting contributions from my project. The first is that TLS can be fairly inefficient. While it is convenient as it happens at a lower level of the network stack, this means that the overhead gets added more often than necessary (since TCP rarely uses more than 500-1000 bytes per packet). Because my player works at the video segment level, it adds overhead much less often. In addition, TLS does encryption and authentication rather than just authentication. This means it will spend extra time and bytes doing cryptography that is not even necessary when all you want to know is if the video came from the correct server. With my player, only authentication is done so nothing extra is spent on encryption. Overall, my project clearly demonstrates that while TLS is extremely convenient, it is not nearly as efficient as it could be.

The second interesting contribution is the importance of a video buffer. The lack of the video buffer was the single biggest problem I encountered during my project. Without this buffer, I was unable to do any kind of processing behind the scenes and instead was forced to do my cryptography between video segments. If the buffer was there, I would have been easily able to hide the delay from the authentication processes instead of making a logical argument that it would work. This is in addition to the fact that the lack of a buffer makes the player particularly susceptible to network delays. All of these just show how important video buffers are to the smooth playback of videos.

8. Extensions

There are two major areas this project could be extended. First, QtSamplePlayer could be updated to have a true video buffer. This would be useful for two reasons. It would help increase performance in general as buffers help prevent pauses in the video at the end of segments. In addition, this would allow me to hide the hash validation delay behind the buffer, so there would be no noticeable pause between larger segments like there currently is in the player. Second, I could create a separate player using TLS and compare the cryptography delay of the two. This would be useful as I was unable to find any good studies of TLS delay to compare my player to. With an implemented TLS authenticated DASH video player, I could definitely see if my player does indeed improve performance.

9. Conclusion

I have created an authenticated DASH video player. By taking advantage of the MPD format of DASH videos, I was able to significantly cut down on the byte overhead when compared to the standard Transport Layer Security authentication. While the lack of a video buffer in QtSamplePlayer caused some problems, the performance of the cryptography was fast enough that there was no noticeable delay in the video. By using my authenticated DASH player, we can more efficiently authenticate videos in the future.

10. Appendix

The following video shows the player successfully authenticating a video, catching an invalid signature, and catching an invalid hash: https://youtu.be/Mrf2zH2rgyA

Here is a list of the major files included with my submission. The locations are in reference to the DashAuthentication folder. The files can additionally be downloaded from github at https://github.com/TKRKS/DashAuthentication/tree/master

- ./CS 6235 Perofmrance.xlsx- An Excel spreadsheet listing all the performance measurements taken for the graphs
- ./fileList- A list of important files and folders in the project (this file has the same information as this appendix).
- ./instructions- Instructions for compiling and running the parts of my project.
- ./mpdModifier- A folder containing all the files related to updating an MPD to include information for authentication
    - ./mpdModifier/Crypto.h- The header file use to include all the cryptography needed for the mpdParser.
    - ./mpdModifier/GenerateKey.cpp- The source code of the generateKey program that creates a public/private key pair.
    - ./mpdModifier/Makefile- The makefile used to compile the mpdParser program.
    - ./mpdModifier/mp4BoxSample- Sample instructions for running the MP4 Box application and making an mpd out of an mp4 file.
    - ./mpdModifier/MpdParser.cpp- The source code for the mpdParser program that updates MPDs to include authentication information.
    - ./mpdModifier/rapidXML- The location of the RapidXML C++ header library.
- ./player- A folder containing all the files related to the video player. This is mainly just the libdash library, as I didn't add any new files to the folder. To find all the files I modified, run the following command in the folder:
  grep -nr "DASH AUTHENTICATION" ./player/
  The most important files that I had to change are listed below:
    - ./player/libdash/libdash/libdash/- The folder with the source for the libdash library.
    - ./player/libdash/libdash/libdash/source/xml/DOMParser.cpp- This file has some of the changes to read the modified MPD's hashes and signature.
    - ./player/libdash/libdash/libdash/source/xml/Node.cpp- This file has some of the changes to read the modified MPD's hashes and signature.
    - ./player/libdash/libdash/qtsampleplayer/- The folder for the video player itself.
    - ./player/libdash/libdash/qtsampleplayer/CMakeLists.txt- The modified CMakeLists.txt that adds Crypto++ and C++11 to the qtsampleplayer compilation.
    - ./player/libdash/libdash/qtsampleplayer/Managers/MultimediaManager.cpp- The file that the changes to authenticate the MPD are located.
    - ./player/libdash/libdash/qtsampleplayer/libdashframework/Input/MediaObject.cpp- The file that the changes to authenticate the video segments are located.
- ./sampleVideo- A folder including a sample MPD and video segments that can be used for testing.

11. References

[1] Apostolopoulos, G., Peris, V., & Saha, D. (1999, March). Transport Layer Security: How much does it really cost?. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (Vol. 2, pp. 717-725). IEEE.

[2] Ferro, G. (2010, March 18). Average IP Packet Size. Retrieved November 27, 2015, from http://etherealmind.com/average-ip-packet-size/

[3] Lederer, S. (2015, April 9). Optimal Segment Length for Adaptive Streaming Formats like MPEG-DASH & HLS. Retrieved November 27, 2015, from http://www.dash-player.com/blog/2015/04/using-the-optimal-segment-length-for-adaptive-streaming-formats-like-mpeg-dash-hls/

[4] 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). ACM.

[5] Williams, O. (2014, November 21). Netflix Now Accounts for 35 Percent of US Internet Traffic. Retrieved September 16, 2015, from http://thenextweb.com/apps/2014/11/21/netflix-now-accounts-35-overall-us-internet-traffic/