Known-Plaintext Attack Against a Permutation Based Video Encryption Algorithm

Adam J. Slagell
slagell@ncsa.uiuc.edu

January 16, 2004

Abstract

One of the approaches to deliver real-time video encryption is to apply permutations to the bytes within a frame of a fully encoded MPEG stream as presented in [2]. We demonstrate that this particular algorithm is vulnerable to a known-plaintext attack, and hence its use should be carefully considered. We also discuss modifications that can make the algorithm resistant to our attack.

1 Introduction

In recent times the demand for multicasting video across distributed systems has grown rapidly. Uses of such technology include video-on-demand, video conferencing, and multicast pay-per-view video. All of these applications have varying degrees of security concerns. For example, military uses may require small but very secure multicast groups for video broadcast or conferencing. Digital cable providers on the other hand may require lighter security; only enough to gain leaker information of pirated video. While computational power has grown steadily, so has the data rate of the video being broadcast over new and faster networks. In turn, this means that the problem of finding robust real-time, software-based encryption solutions for multicast video has not been solved.

There are two main classes of encryption solutions: those which act on fully encoded MPEG streams and those which work on partially decoded MPEG streams. The algorithm [2] we analyze works on the fully encoded stream. To be effective when working on the entire stream, the algorithm must be extremely fast (For example, to achieve the frame rate of NTSC video, each frame must be decoded and decrypted in 33 ms. On modern machines it is not even possible to do just the decoding of HDTV frames that quickly in software). The algorithm discussed is in fact very fast and appears to be a good solution for real-time video encryption. However, we will demonstrate that the algorithm is vulnerable to a known-plaintext attack, and as such one should be cautious and aware of the vulnerabilities of this algorithm before using it. Specifically, it must be avoided when the contents of any of the frames are known, since we show that in most cases only one frame of known-plaintext is needed to recover the key. However, we do discuss some improvements with minimal performance impact that make the algorithm resistant to the attack we present here.

This paper is organized as follows: Section 2 discusses many current solutions to video encryption, focused mainly around MPEG video. Section 3 explains, in detail, the algorithm used in [2] for video encryption. Section 4 explains and demonstrates our plaintext attack upon the algorithm

in [2]. Section 5 discusses some modifications to improve the system implemented in [2]. Section 6 concludes this paper.

2 Related Work

There are two main classes of video encryption solutions: those which act on fully encrypted MPEG streams and those which work on partially decoded MPEG streams. The most straightforward solution is to simply encrypt the entire data stream with a general purpose symmetric key algorithm. This is often referred to as the *naive solution*. The advantage of this method is not only simplicity of implementation, but it allows modularity of code in a video multicasting system. A good example of this is a framework developed by Prabhu in [6]. This framework allows one to easily change encryption algorithms, key distribution systems and even video encoders/decoders. It is essential that decryption and decoding not be intermingled for this to work. However, a clear disadvantage is computational overhead to do cryptographic operations. In a system that uses hardware for decoding, a software decryption solution could be a bottleneck.

The Video Encryption Algorithm (VEA) developed by Qiao and Nahrstedt in [7] also falls into the same category as it applies encryption to the fully encoded video stream. However, VEA depends upon statistical properties of MPEG and uses a standard symmetric algorithm to reduce the amount of data that is actually encrypted. This can yield an almost 50% gain in performance over the naive solution.

An even faster encryption algorithm was developed by Chu, Qiao and Nahrstedt in [2]. This is the algorithm we describe in detail in section 3 and describe a known-plaintext attack against it in section 4. This algorithm works on a frame by frame basis, permuting elements on a byte level of granularity. This permutation is of course much simpler than the complex operations of a full-fledged block cipher.

Another approach is to selectively encrypt the I-frames of the MPEG stream. This falls into the second category since the MPEG stream is only partially encoded when encryption is implemented. This selective algorithm is discussed in ([3],[4]). However, in [1] Agi and Gong show this method is insufficient if we desire the video to be completely unviewable. This attack is possible by exploiting the inter-frame correlation and using the unencrypted I-blocks found in P and B frames.

SECMPEG, developed by Meyer and Gadegast in [5], is a complete rework of the MPEG format with security in mind. It uses selective encryption and has additional headers thus making it incompatible with standard MPEG encoders and decoders.

The ZigZag-Permutation algorithm in [9] is also a selective encryption algorithm. The idea here is to use a permutation of the zig-zag ordering of DC and AC coefficients in the DCT transformation of 8x8 blocks in the MPEG or JPEG format. However, in [8] Qiao shows that the ZigZag Permutation algorithm is vulnerable to a known-plaintext attack as well as significantly increases the size of MPEG files.

3 Permutation Real-time MPEG Encryption Algorithm

Here we describe the permutation based encryption algorithm of [2]. It is extremely fast and useful in situations where encryption speed is the bottleneck. Such an example would be when hardware decodes the video, but decryption must be done in software. It is also intended mostly for short

time use where the data loses value quickly. An example could be encrypting news stories that need only be protected for a matter of hours or a few days.

The main idea is that one uses the key to determine how to distribute the bytes of the video frame into several blocks of data whose sum capacity is that of the original frame. So the first step is to determine the size of each of the blocks from the key and the size of the frame. Then the bytes of the original video frame are interleaved into these different blocks. One can think of each of these blocks as bins, and the algorithm simply runs through the original frame, byte by byte, dropping one byte into each bin. It continues to cycle through all of the bins until they are full. Some bins will fill up faster than others since the sizes are different. Figures 1 and 2 show an example of a 100 byte video frame being encrypted (Note: this example is of reduced size to illustrate the algorithm. Actual data may be 2 orders of magnitude larger.) The process may actually involve an extra step because the sum capacity of all the blocks may be slightly less than the total size of the original frame. In this case there will be a few bytes leftover after all the bins have been filled. We place these in what we call the "leftover" bin.

26	67	54	07	CB	24	$9D$	17	86	44
AB	$F5$	$E1$	$3D$	$7E$	DD	$B4$	$C4$	FE	BA
72	27	$9D$	EF	39	$0A$	69	81	48	$6F$
42	CB	$2D$	39	$7F$	AD	15	$B8$	$C2$	$3D$
$5F$	$8F$	$6E$	$2B$	24	61	58	$D3$	FD	$6D$
51	89	$A5$	05	BE	87	$E4$	05	30	56
09	18	10	$3D$	DE	00	57	FB	56	78
73	43	77	$4F$	49	44	50	91	21	FB
$3E$	44	25	$D0$	29	86	$9A$	49	$3A$	$8B$
$D7$	$2C$	$D2$	56	$1C$	$8B$	44	11	$D4$	21

Figure 1: 100 byte sample video frame. Read left to right, top to bottom.

We use the example from [2] to illustrate how the key determines bin sizes. Let the frame size be 5000 bytes and the encryption key be 5603417982. The first step is to sum the digits of this decimal number. In this case the sum is $45 = 5 + 6 + 0 + 3 + 4 + 1 + 7 + 9 + 8 + 2$. The number of bins is the same as the number of digits, in this case 10. The size of the i^{th} bin is a proportion of the total number of bytes. Explicitly, this proportion is defined as the value of the i^{th} digit of the key divided by the sum found in the first step. This proportion of the total number of bytes will of course not come out evenly. So the numbers we calculate are truncated. Below we have the bin sizes calculated for the key **5603417982**.

Block 1: $\lfloor \frac{5}{45} \cdot 5000 \rfloor = 555$ B
Block 2: $\lfloor \frac{6}{45} \cdot 5000 \rfloor = 666$ B
Block 3: $\lfloor \frac{0}{45} \cdot 5000 \rfloor = 0$ B
Block 4: $\lfloor \frac{3}{45} \cdot 5000 \rfloor = 333$ B
Block 5: $\lfloor \frac{4}{45} \cdot 5000 \rfloor = 444$ B
Block 6: $\lfloor \frac{1}{45} \cdot 5000 \rfloor = 111$ B
Block 7: $\lfloor \frac{7}{45} \cdot 5000 \rfloor = 777$ B
Block 8: $\lfloor \frac{9}{45} \cdot 5000 \rfloor = 1000$ B
Block 9: $\lfloor \frac{8}{45} \cdot 5000 \rfloor = 888$ B

			21	
			$D4$	
			11	
			44	
	$1C$		$8B$	
	$D2$		56	
	$D7$		$2C$	
	$3A$		$8B$	
86	$9A$		49	
25	$D0$		29	
FB	$3E$		44	
50	91		21	
77	$4F$	49	44	
56	78	73	43	
DE	00	57	FB	
09	18	10	$3D$	
87	$E4$	05	30	56
51	89	$A5$	05	BE
61	58	$D3$	FD	$6D$
$5F$	$8F$	$6E$	$2B$	24
AD	15	$B8$	$C2$	$3D$
42	CB	$2D$	39	$7F$
$0A$	69	81	48	$6F$
72	27	$9D$	EF	39
DD	$B4$	$C4$	FE	BA
AB	$F5$	$E1$	$3D$	$7E$
24	$9D$	17	86	44
26	67	54	07	CB

Figure 2: Sample frame deposited into bins in a round-robin manner. Key is 56473.

Block 10: $\lfloor \frac{2}{45} \cdot 5000 \rfloor = 222$ B

There are a couple of things to notice. First, if there is a 0 in the key, it really does nothing. We could add or remove 0's to the key without any effect. The second thing to notice is there is a maximum on the number of leftover bytes. Each truncation can cause no more than one byte to be leftover in the end. So the number of digits is an upper limit on the number of leftover bytes. In this case it is 10. The actual number of leftover bytes can be calculated to be 5 in this case. So Block 11 has size 5 bytes.

4 Plaintext Attack

Our attack is a known-plaintext attack which means that some of the unencrypted data must be available. In our case, one unencrypted frame must be known. This could realistically happen if we know the beginning is completely black for a few frames or if we know that all videos from this creator begin the same. An example might be the roaring lion at the beginning of MGM

26	24	AB	DD	72	$0A$	42	AD	$5F$	61
51	87	09	DE	56	77	50	FB	25	86
67	$9D$	$F5$	$B4$	27	69	CB	15	$8F$	58
89	$E4$	18	00	78	$4F$	91	$3E$	$D0$	$9A$
$3A$	$D7$	$D2$	$1C$	54	17	$E1$	$C4$	$9D$	81
$2D$	$B8$	$6E$	$D3$	$A5$	05	10	57	73	49
07	86	$3D$	FE	EF	48	39	$C2$	$2B$	FD
05	30	$3D$	FB	43	44	21	44	29	49
$8B$	$2C$	56	$8B$	44	11	$D4$	21	CB	44
$7E$	BA	39	$6F$	$7F$	$3D$	24	$6D$	BE	56

Figure 3: Ciphertext frame. Read bytes from bottom to top of bin, from left bin to right bin. Key is 56473

P_1	P_6	P_{11}	P_{16}	P_{21}	P_{26}	P_{31}	P_{36}	P_{41}	P_{46}
P_{51}	P_{56}	P_{61}	P_{65}	P_{69}	P_{73}	P_{77}	P_{80}	P_{83}	P_{86}
P_2	P_7	P_{12}	P_{17}	P_{22}	P_{27}	P_{32}	P_{37}	P_{42}	P_{47}
P_{52}	P_{57}	P_{62}	P_{66}	P_{70}	P_{74}	P_{78}	P_{81}	P_{84}	P_{87}
P_{89}	P_{91}	P_{93}	P_{95}	P_3	P_8	P_{13}	P_{18}	P_{23}	P_{28}
P_{33}	P_{38}	P_{43}	P_{48}	P_{53}	P_{58}	P_{63}	P_{67}	P_{71}	P_{75}
P_4	P_9	P_{14}	P_{19}	P_{24}	P_{29}	P_{34}	P_{39}	P_{44}	P_{49}
P_{54}	P_{59}	P_{64}	P_{68}	P_{72}	P_{76}	P_{79}	P_{82}	P_{85}	P_{88}
P_{90}	P_{92}	P_{94}	P_{96}	P_{97}	P_{98}	P_{99}	P_{100}	P_5	P_{10}
P_{15}	P_{20}	P_{25}	P_{30}	P_{35}	P_{40}	P_{45}	P_{50}	P_{55}	P_{60}

Figure 4: Cipher text viewed in a manner to show how original bytes have been permuted. Key is 56473

films. Using the unencrypted and encrypted version of a frame, we are able to retrieve the key used to encrypt that frame. This allows us to decrypt the rest of video encrypted with that key. The effectiveness of our attack of course depends upon the method and frequency of rekeying operations.

4.1 A Periodic Pattern

The key to attacking this algorithm is to exploit the periodic structure resulting from how the data is interleaved between these blocks. We use the examples from figures 1 and 2 to demonstrate the pattern and attack. These examples use a 100 byte frame with the key 56473. We will use the notation P_i to denote the i^{th} byte of plaintext and C_i to denote the i^{th} byte of ciphertext. Figures 3 and 4 show the encrypted frame, but in different ways. Figure 3 shows the actual byte values, and figure 4 shows the corresponding place the bytes have in the plaintext stream, thus illustrating exactly how the bytes have been permuted.

Examine the beginning (bottom) of the first bin. The first byte is the first byte of the original frame, i.e P_1. The next byte is P_{n+1}, where n is the number of non-empty bins, i.e. the number of nonzero digits in the key. In our example, $n = 5$, and we find that the first two bytes of the ciphertext are P_1 and P_6. The next byte of ciphertext is $P_{2n+1} = P_{11}$. What we see is that there is a period of n bytes in the beginning of this bin. In fact, it should become clear that this happens

at the beginning of every bin, figure 2 supporting this claim.

Going further with this, we see the following: starting at the first bin there is a period of n, then $n-1$, then $n-2$ and so on. When we reach bin two, the period will jump back to n. (Note, in fact, it does not necessarily drop by only one. If k bins have the same size, they all fill at the same time causing the period to drop by k). In our example, the period drops to 4 after C_{13} and drops again to 3 after C_{17}. The period jumps back to 5 after C_{21}. This can be verified in figure 4.

This monotone decreasing behavior of the period followed by the jump back to n will happen at the boundary of every bin except in one special case. It could be that two or more bins of the minimum size are adjacent. This corresponds to two more of the minimum digits in the key being adjacent. If this is the case, the period will be the same across this series of bins, but there will be discontinuities between them. To illustrate this, let the first byte of the first of the two adjacent be P_i and the last byte of this bin be P_j. Then at the boundary point there will be a discontinuity in the pattern because P_j will not be followed by the P_{j+n} in the ciphertext. Instead, P_j will be followed by P_{i+1}.

4.2 Exploiting the pattern

The pattern above gives us a way to find the boundaries between the bins. Since we know the original frame, it will not be difficult to find the period of the first bin. We know the first byte. We look at the next byte in the encoded stream, C_2, and check if $C_2 = P_k$, where k is our guess at the key length. It is not actually necessary to know k, but it just speeds things up a bit if we have a good estimate. Key length usually isn't secret, but if it is secret it will take at most $O(k)$ operations where k is the actual key length. Since the key length is much smaller than the size of the video frame, this small overhead is of little concern.

As noted in [2], the frequency of any byte value in a fully compressed MPEG frame is about 1 in every 256. In other words, all byte values are equally likely. So chances are that if $C_2 = x$, and n is the smallest integer such that $x = P_{n+1}$, then the period is n. In our example $x = 24$, and we do find that the smallest integer such that $P_{n+1} = 24$ is 5. This is consistent with the true period of the first bin. In general, we can test our hypothesis by using the assumption to predict the next byte in the encrypted frame. In our example the hypothesis would be that the period is 5, and we would predict that $C_3 = P_{2n+1} = P_{11}$. And we would find our prediction to be correct in the example. If we successfully predict 5 or 6 bytes in the encrypted stream, the odds are very strong that we have the correct period. This fact follows since the probability of a byte having a particular value is about $\frac{1}{256}$. If we found that using period n did not predict correct results very long, we go to the next byte of value x in the original frame. If it is P_{m+1}, we would guess the period to be m. We test this new hypothesis in the same manner. It will not take long to find period in this manner, especially since the key size is small compared to the size of the frame.

Eventually, we will find that after correctly predicting many values of the ciphertext, we will have a misprediction. In our example this first happens at C_{14}. This is because, as we discussed earlier, the period will decrease. In our example it decreased to 4. So we try smaller and smaller values of the period until we start getting good predictions again. This will keep happening until we hit the next bin. In our example this happens at C_{21}. Then, decreasing the period will not work. At this point we check if the next byte is equal to byte two of the original frame. (In our example, this is exactly what happens as we notice $C_{21} = P_2$). If we have actually found the beginning of bin 2, we know the value should equal P_2. We know that we have in fact found the beginning of

the next bin because our prediction has suddenly gone bad after going well for so long.

It is possible, but improbable, that we will make good predictions past the actual bin boundary and hence not see the discontinuity. For example, our prediction for the byte of ciphertext at the beginning of the second bin may coincidentally be equal to the value of P_2. But if this happens, we will have difficulties locking onto a period of n at what we believe to be the beginning of the next bin, i.e. where we first notice a discontinuity. So in such a case we back up one byte and test if that is the beginning of the second bin, i.e. has a value equal to P_2. It is highly improbable that we will make several correct predictions past the actual bin boundary, and hence not see a discontinuity immediately. Thus we will not have to back up very far, if at all.

We can continue in this manner to find the rest of the bin boundaries. Only note that if we have two adjacent bins of minimum size, we will never see the period drop below the max period between the two bins. Instead, we will suddenly have bad predictions to indicate that we have reached a bin boundary. In the last few bytes we may have some more bad predictions. This will be the leftover bin. We can verify this by comparing these last few bytes of the encrypted stream with the last few bytes of the original frame.

4.3 Putting It All Together

Once we discover the bin sizes, we can compute the proportion of the bin size to the frame size. Now we know this fraction for the i^{th} bin to be almost equal to $\frac{d_i}{s}$ where d_i is the i^{th} digit and s is the sum of all the digits. Our goal is to find a common denominator for all these fractions, such that all the numerators are between 1 and 9. Actually, these numerators will not necessarily be integers because of the truncation, but they will be an integer or just slightly less than an integer. There are few denominators that will keep all the numerators in the range. The most likely case in which we could get multiple denominators that keep the numerators within the range is if the key consists entirely of 1's. However, we have the additional condition that these numerators are integers or just slightly less than an integer. All in all, there will be few, and probably only one, candidate for the key. If we have a couple possible key values, we can try them on subsequent frames to verify which one is correct. Once we have all the numerators, we simply round up to find the integer value of each digit of the key.

In our example we see that bin one has 20 bytes, bin two 24, bin three 16, bin four 28, and bin five 12. This yields the corresponding fractions: $\frac{20}{100} = \frac{1}{5}$, $\frac{24}{100} = \frac{6}{25}$, $\frac{16}{100} = \frac{4}{25}$, $\frac{28}{100} = \frac{7}{25}$, and $\frac{12}{100} = \frac{3}{25}$. Solving for a common denominator we get: $\frac{5}{25}$, $\frac{6}{25}$, $\frac{4}{25}$, $\frac{7}{25}$, and $\frac{3}{25}$. Notice that this is the **only** common denominator that produces numerators that are all single digits. Thus we can conclude the key is 56473.

Notice that this method will never yield a key with zeros. It gives the unique key without zeros. As mentioned above, we can add zeros anywhere in the key without effect. The information of the zeros is lost when doing the encryption, but it was unimportant in the first place. It is also interesting to notice that we only need one frame of plaintext to perform this attack. Of course, it being a probabilistic algorithm, there is always a small possibility of failure. In such cases this algorithm could be run on another frame. Furthermore, this algorithm runs in linear time with the constant factor being largely dependent upon the number of steps we are willing to back up if there is a misprediction and the number of matches we require before concluding we lock onto the period. But these constants can be very small since the probability of guessing n bytes correctly is $\frac{1}{256^n} = 2^{-8n}$.

5 When is the Permutation Algorithm Acceptable?

It is not improbable that the first frame of a video is known. MGM has the clip of the roaring lion, Warner Brothers has their unique clip with the WB studios and their emblem, and Dreamworks has the clip with the water and their logo. So certainly, an attacker may have known-plaintext at the beginning of a video. If the same key is used to encrypt the entire video and this beginning portion, then the attacker can decrypt the entire video.

An easy solution is to make it less likely for the attacker to have plaintext. We can simply leave the introduction unencrypted. Most people will find it acceptable for an attacker to know what company made the video. In fact, the attacker may know the exact video she is trying to obtain and is not trying to decrypt a video of unknown content. However, this solution only reduces the odds of finding known-plaintext. It is still that possible the attacker will have part of a video's plaintext, but not the entire video. Perhaps she got it from a sample or trailer.

We would like a solution that does not assume the attacker is unable to obtain plaintext. The most obvious solution is to frequently re-key. Now we must be careful how the key is exchanged. If the new key is appended to the end of a frame, then decrypting the current frame will give us the key to the next frame. This is certainly not an improvement. In fact, most of the key may be inside the leftover bin. So just looking at the end of an encrypted frame gives the end of the key in plaintext!

One might think that a good alternative is to put the key in the beginning of the frame. She may think this makes the attack more difficult, but it will simply make the first byte of each bin incorrect. The above algorithm only needs to be modified slightly. If we additionally pad the key with worthless zeros, the first *few* bytes of some bins will be part of the key and not the video. In this case the algorithm would need to be modified a bit more. It would need to take care of the possibility that the first bin will not have the first byte of the video in the first byte of the bin. Instead, one of the early bytes of the unencrypted frame will be a few bytes into the first bin. This adds a little more trial and error into the algorithm, but eventually it should lock onto the period and work fine.

The simplest way to handle this may be to use an outside method of distribution for re-keying. This could be through a separate means of communication, or one could use a previously agreed upon secret key and algorithm to encrypt the new permutation algorithm key. Then this separately encrypted key could sent every so often with the video stream to enable re-keying. However, there are significant costs to setting up a key distribution system and performing many re-keys, and thus we wish to examine other solutions as well.

Another solution one may come up with to thwart this attack is to use a very large key. Using a key 20% of the size of the frame, means the average bin size is 5 bytes. With bins this small, we cannot reliably lock onto the period. The obvious drawbacks are the extra data sent if re-keying is frequent and the extra memory usage for maintaining state in the decryption algorithm.

Lastly, one can modify the algorithm with the specific goal of thwarting this particular type of attack. A key factor to the periodic behavior noticed in the pattern is that the algorithm simply puts bytes into bins in a round-robin fashion. By skipping over bins by a number of bins that does not divide the total number of bins, we can break the observed pattern (Though this does not guarantee that we do not still have patterns exploitable by more sophisticated attacks). For instance, we could put a byte into every p^{th} bin, where p is a prime larger than the total number of bins. This will disturb the recognized pattern significantly. Perhaps even better, we could use

a recursively defined sequence taken modulo the number of bins to determine into which bin to place the current byte. For example, we could use the Fibonacci numbers. And if we really would like to break up any possible patterns, we could use information from a Pseudo-Random Number Generator (PRNG) to determine when to skip over bins. The PRNG could be seeded by another shared secret or some derivation of the encryption key which must already be shared. This would make the algorithm bear more of a resemblance to a stream cipher.

Of course, any of these modifications, except for skipping bins by a fixed number, will slow down the encryption and decryption processes. In the case of using a recurrence relation, it depends upon the complexity of the recurrence. Using the Fibonacci sequence would only require one more addition at each step. One addition is already required to increment the bin even if only by one. So this would not affect the Big-Theta running time of the algorithm. Just as there different recurrences that could take varying amounts of computation, there are many PRNGs. Time complexity would need to be evaluated on an individual basis, but it is safe to conclude that any PRNG is going to be significantly more complicated than computing a simple recurrence. Also more extensive analysis would be required by the cryptography community to determine how effective these measures are since they are only guaranteed to disrupt the pattern we observed. Simple solutions could easily create different and more complex patterns that are still exploitable.

6 Conclusion

In this paper we discussed the vulnerabilities of the permutation video encryption algorithm in [2]. This is a very fast encryption algorithm that works on a fully encoded MPEG stream. The key is used to split the frame into blocks in which bytes are deposited in a round-robin fashion. In section 4 we demonstrated a powerful known-plaintext attack against this algorithm.

Finally, we discussed some modifications that could make this algorithm a viable solution. We also discussed some modifications which appear at first to help, but in fact do little to add security. We conclude that frequent re-keying and an outside method of distributing the new keys provides reasonable security while still taking advantage of the fast encryption/decryption rates, but this comes at the cost of a more complicated key distribution system. Mostly, such a solution limits the amount of damage a frame of leaked plaintext can cause.

7 Acknowledgments

I would like to thank Dr. Klara Nahrstedt for her invaluable help in writing my first professional paper. While I have co-authored a paper published before this one, this was written a year and a half before submission to the ePrint archives. I would also like to thank Himanshu Khurana, Jim Basney, Rafael Bonilla and Bill Yurcik for their comments and suggestions.

References

[1] I. Agi and L. Gong. An empirical study of secure MPEG video transmission. In *IEEE Symposium on Network and Distributed Systems Security*, 1996.

[2] Hao-hua Chu, Lintian Qiao, and Klara Nahrstedt. A secure multicast protocol with copyright protection. In *Proceedings of IS&T/SPIE's Symposium on Electronic Imaging: Science and Technology*, 1999.

[3] Y. Li, Z. Chen, S. Tan, and R. Campbell. Security enhanced MPEG player. In *Proceedings of IEEE First International Workshop on Multimedia Software Development (MMSD'96)*, Berlin, Germany, March 1996.

[4] T.B. Maples and G.A. Spanos. Performance study of MPEG video transmissions. In *Proceedings of the 4th International Conference on Computer Communications and Networks*, Las Vegas, Nevada, 1995.

[5] J. Meyer and F. Gadegast. Security mechanisms for multimedia data with the example MPEG-1 video. In *Project Description of SECMPEG, Technical University of Berlin*, 1995.

[6] Raghavendra Prabhu. Distributed security framework for multimedia transmission. Master's thesis, University of Illinois, 2000.

[7] L. Qiao and K. Nahrstedt. A new algorithm for MPEG video encryption. In *Proceedings of The First International Conference on Imaging Science, Systems, and Technology (CISST'97)*, pages 21–29, Las Vegas, Nevada, July 1997.

[8] L. Qiao, K. Nahrstedt, and I. Tam. Is MPEG encryption by using random list instead of zigzag order secure? In *IEEE International Symposium on Consumer Electronics*, Singapore, December 1997.

[9] Lei Tang. Methods for encrypting and decrypting MPEG video data efficiently. In *ACM Multimedia*, pages 219–229, 1996.