Steal This Movie - Automatically Bypassing DRM Protection in Streaming Media Services

Ruoyu Wang[1,2], Yan Shoshitaishvili[1], Christopher Kruegel[1], and Giovanni Vigna[1]

[1]University of California, Santa Barbara, CA, USA
{fish,yans,chris,vigna}@cs.ucsb.edu
[2]Network Sciences and Cyberspace, Tsinghua University, Beijing, China

Abstract

Streaming movies online is quickly becoming the way in which users access video entertainment. This has been powered by the ubiquitous presence of the Internet and the availability of a number of hardware platforms that make access to movies convenient. Often, video-on-demand services use a digital rights management system to prevent the user from duplicating videos because much of the economic model of video stream services relies on the fact that the videos cannot easily be saved to permanent storage and (illegally) shared with other customers. In this paper, we introduce a general memory-based approach that circumvents the protections deployed by popular video-on-demand providers. We apply our approach to four different examples of streaming services: Amazon Instant Video, Hulu, Spotify, and Netflix and we demonstrate that, by using our technique, it is possible to break DRM protection in a semi-automated way.

1 Introduction

Digital Rights Management (DRM) is used by content distributors to restrict the way in which content may be used, transferred, and stored by users. This is done for several reasons. To begin with, content creators try to prevent content from reaching non-paying users through pirated copies of the content. While estimates of the cost impact of piracy are considered to be hugely inaccurate and research on this issue is inconclusive [41], they vary from $446 million [14] to $250 billion [8] for the movie and music industries in the US alone, and are far from insignificant in other parts of the world [12, 27]. Consequently, DRM is used to protect the media distributed through subscription-based services. In these services, such as Netflix, Spotify, Hulu, and Amazon Prime Instant Video, a user pays a recurring fee for access to a large database of media. This media can be played as much and as often as the user wishes, but becomes unavailable when a user stops paying for the service. The need to protect content in this scenario is obvious: if users can save the content for playback later and simply cancel their account, the streaming service will lose substantial amounts of money.

DRM protection of media, especially passive media such as movies and music, has a fundamental difficulty. In order to enable the viewing of content, such content must at some point be decrypted. Different DRM schemes put this decryption at various stages of the media playback pipeline. Schemes such as High-bandwidth Digital Copy Protection (HDCP) [9] attempt to put this decryption outside of the reach of software and into the media playback hardware itself. However, use of these schemes are not always feasible. Specifically, many mobile devices, virtual machines, and lower-end computers do not support schemes such as HDCP. To function on such devices, DRM schemes must carry out decryption in software. On top of this limitation, hardware DRM schemes suffer from a problem of being too brittle against attacks. This was demonstrated, in the case of HDCP, with the compromise of the HDCP master keys [30], which rendered that DRM scheme useless. DRM schemes that do not rely on special hardware support are much more flexible in recovering from such compromises.

In order for an effective DRM scheme to be implemented, the possible attacks that it could succumb to must be well understood. In this paper, our goal is to examine one such attack: the identification of the transition between encrypted and decrypted data in the media player software.

To this end, we introduce MovieStealer, an approach for the automatic defeating of DRM in media playing programs. This approach takes advantage of several central intuitions. Our first intuition is that most data-processing operations, and specifically decryption operations, are carried out on buffers of data. This allows us to concentrate our analysis on the flow of data between buffers, making the analysis task considerably more tractable. Secondly, we observe that all popular media services of which we are aware utilize existing media codecs. We believe that this is because coming up with new codecs is a very complicated task, and many of the technologies behind efficient codecs

are patented. Additionally, high-definition codecs are extremely performance-intensive, and many media player devices rely on hardware support to decode them. This reliance on hardware support makes changing these codecs extremely difficult, making it far easier to license an existing codec than to create a new one. Utilizing this observation, we are able to identify buffers that contain data similar to what we would expect to be present in popular codecs. Our final observation is that we can distinguish three distinct classes of data by carrying out a statistical analysis: encrypted data (which will possess high randomness and high entropy), encoded media data (which will possess low randomness and high entropy), and other data (which will possess lower randomness and entropy).

We utilize these observations and develop an approach that tracks the flow of data from buffer to buffer within a program and identifies, using information theoretical techniques, the point in the program at which the data is decrypted. After automatically identifying this location in the program, MovieStealer dumps the decrypted stream. This stream can then be reconstructed into an unprotected media file and played back in an unauthorized media player.

Furthermore, we design optimizations that allow this online approach to be carried out on a running media player. Such optimizations are necessary due to the performance-demanding nature of the services that we target.

We implemented this approach and evaluated it on several streaming services, namely Netflix, Amazon Instant Video, Hulu, and Spotify. The latter is a music streaming service, while the others are video streaming services. All of these services are real-time, high-performance products which must be analyzed with low overhead in order to function. In all cases, MovieStealer is able to successfully pinpoint the decryption location and dump the decrypted stream. After this point, we consider the DRM protection to be *broken*. We have also implemented media file reconstructors to recover a playable media file.

To showcase our optimizations, we have also evaluated our approach against GPG, an open-source cryptographic suite.

The task of dumping the decrypted stream is completely automated. MovieStealer dynamically analyzes a program while it is used to play media, and dumps the decrypted streams. However, the final step of reconstruction requires a component to be developed for each protocol. We have implemented three such components to cover our four target streaming services. Since we consider the DRM to be bypassed as soon as we recover the decrypted data, automating this last step is out of the scope of our DRM analysis.

MovieStealer was developed in order to gain insight into the weaknesses of cryptographic DRM schemes. The implementation and utilization of such an approach for piracy purposes is, of course, illegal. Our intention is not to aid illegal activity, and we present a discussion on ethics and legality in Section 7.

In summary, we make the following contributions:

1. We present an approach capable of automatically identifying and exploiting weaknesses in DRM scheme implementations by identifying cryptographic functionality in real-time, with no offline analysis, and duplicating the decrypted data.
2. To make such an approach work on performance-demanding applications and to reduce the amount of time the approach requires to locate the decrypted data, we utilize a set of optimizations that would be useful for any similar dynamic analysis approaches.
3. We show the effectiveness of this approach on four popular streaming services (Amazon Instant Video, Hulu, Netflix, and Spotify) and a general-purpose encryption tool (GPG).
4. To the best of our knowledge, we demonstrate the first publicly-described approach to duplicate PlayReady-protected content (such as modern versions of Netflix) without the use of a screen scraper. While we have been informed that there have been other attacks on PlayReady, we have been unable to find any public evidence of this fact.
5. Finally, we suggest several countermeasures that vendors of content protection schemes could employ to resist an attack such as MovieStealer. These range from technical solutions, attacking the technical details of our approach, to social solutions, such as increased use of watermarking to make piracy more prosecutable.

2 Background and Related Work

Over the last several decades, there has been an arms race between content owners, wishing to restrict the use of their content, and content consumers, who wish to use such content in an unrestricted way. New Digital Rights Management techniques are created on a regular basis, and new workarounds are quickly found to counter them. In this section, we survey several popular DRM techniques to better frame the research presented in this paper.

DRM schemes can generally be split into two classes: non-cryptographic DRM schemes and cryptographic DRM schemes. The former relies on verifying that the user is authorized to use the protected content by somehow utilizing a physical aspect of this content. Of course, this requires that the content ships with something like a manual, disk, or hardware dongle to use for verification. With the advent of digital distribution for software and multimedia, non-cryptographic DRM schemes have fallen in popularity.

On the other hand, cryptographic DRM schemes work by cryptographically verifying that the user attempting to access the content is authorized to do so. This approach is usable for digital distribution of content, and

is the paradigm according to which modern DRM schemes are developed.

In this paper, we include *link-protection* schemes, such as HDCP, which protect content in transit from being intercepted, with true *Digital Rights Management* systems, which ensure that only an authorized user is accessing the content in question. From the viewpoint of removing the protection, these two categories of content protection schemes are quite similar, and our system is general enough to handle both.

2.1 Cryptographic DRM Techniques

One of the early examples of cryptographic DRM techniques was the DVD Content Scramble System [44]. CSS is an encryption scheme to prevent DVDs from being played on unauthorized devices. It functioned by assigning a limited number of keys to manufacturers of DVD playing devices. These keys would then be used to decrypt the key chain of a given DVD and play back the video. CSS was broken in 1999 through cryptanalysis by a group of security researchers including Jon Lech Johansen (afterwards known as DVD Jon) [48]. This was done by reverse engineering a software DVD player to identify the encryption algorithm.

CSS was a forerunner of the type of copy protection that MovieStealer was created to analyze. While DRM schemes have since evolved to be more flexible, the basic premise remains the same: content is shipped in an encrypted form (whether through physical media or as a download), and is decrypted by an authorized player.

2.2 Hardware-based DRM

Hardware-based DRM has been around since the early days of copy protection. Early examples of this class of approaches are copy-protection dongles shipped with software [49]. Software protected by such dongles does not run without the presence of the dongle, and duplication of the dongle is infeasible. While early dongles simply contained static information that would be checked in software, modern dongles are complex cryptographic co-processors that actually carry out operations, such as decrypting the program code, on behalf of the protected program.

A specific adaptation of this into the realm of multimedia is HDCP [9], a link protection scheme which moves the decryption of media content outside of the computer. In a perfect implementation of this scheme, all content handled by the computer is always encrypted [11], and the decryption occurs in the media playback hardware (such as the monitor) itself. This would be problematic for our approach, but is not a problem in practice for several reasons. To begin with, all of our surveyed streaming services allow playback without HDCP. This is necessary because systems such as netbooks and virtual machines lack support for HDCP, and these services attempt to remain compatible with them. Additionally, HDCP does not integrate seamlessly with the encryption used in the media streaming services of which we are aware. Encrypted content streamed from these services must first be decrypted, usually in memory, before being re-encrypted with HDCP. While some media devices exist that can handle this step in dedicated hardware, thus disallowing any access to the unencrypted stream, general purpose consumer devices are not among them. This means that on such devices, even in the presence of HDCP, MovieStealer can intercept the protected content on such devices while it is unencrypted. Finally, HDCP has been irrevocably broken with the leak of the HDCP master key. Hardware-based DRM schemes like HDCP are very hard to patch because they need to work on many devices that are not easily upgradeable. While the upgradeability of these devices might be improved in the future, there is currently no clear solution to this issue.

	Encryption type		
DRM Platform	Connection	File	Stream
PlayReady	No	No	Yes
RTMPE	Yes	No	No
Spotify	Yes	Yes	No

Table 1: The present encryption locations for our analyzed platforms.

2.3 Streaming DRM Platforms

We analyze three different DRM schemes used by four platforms in this paper: Microsoft PlayReady (used by Netflix) [10], RTMPE (a link protection mechanism used by Adobe's Flash streaming platforms such as Amazon Instant Video and Hulu) [3], and Spotify's content protection [15].

We stress that our approach, as implemented by MovieStealer, does not exploit any particular vulnerability inherent to any single platform. Instead, these DRM schemes are vulnerable due to their inherent design, and not the inadequacies of any specific vendor or organization.

In this section, we provide some details about how these schemes function, in order to better frame our approach.

2.3.1 PlayReady

Microsoft's PlayReady DRM, as implemented in its Silverlight streaming platform, which is used most prominently by Netflix, is a cross-platform content protection mechanism. PlayReady supports *individualization*, meaning that the media is encrypted with a content key, which is then encrypted with different keys for every user. Every time a user streams content on Silverlight, PlayReady provides an individualized *license*, ensuring that the content key can be decrypted and protected content viewed only by the intended recipient. The process to play back PlayReady-protected media using Silverlight comprises

several steps. To improve understanding, we present a high-level overview of these steps.

Metadata. To initialize playback, the Silverlight client requests metadata from the media server provider (such as Netflix). This metadata is a file that contains available resolutions and bit rates for the content, whether the payload is encrypted or not, the domain name of the license server, and the expiration time of the request.

License. If the metadata specifies that the payload is encrypted, the Silverlight client must acquire the license (containing the decryption key) from the license server, which is specified in the metadata. When a client sends the license request to the license server, the license server responds with the *Individualized Black Box* (IBX). The IBX is a custom, easily-upgradeable, and highly-obfuscated DLL that can be customized by individual content providers. Using the IBX, the client generates an individualized request to the license server.

The license server verifies this request and responds with a license. The client uses the IBX to decrypt the license and extract the content key, which is a 128-bit AES key.

Data. Having acquired the license, the client can now play back the protected content. This content takes the form of a fragmented MPEG-4 file transferred from the service provider. The protection works by encrypting the media stream data, while leaving the headers and stream metadata unencrypted. The data is encrypted using AES and is decrypted using the key acquired from the license server.

Performance. PlayReady has several performance requirements. To begin with, as with any network service, the client must be able to communicate with the server without letting the connection time out. Additionally, as a security measure against piracy, the IBX and corresponding license request have an expiration time, and the license will stop working after this timeout has elapsed. Finally, the media player (Netflix) itself has a minimum performance threshold, below which it will stop processing the stream and display an error. A successful online analysis of a PlayReady-protected media player must have a low-enough overhead to allow the player to meet these performance obligations.

2.3.2 RTMPE

RTMPE is a lightweight link protection mechanism developed by Adobe on top of the Real Time Messaging Protocol (RTMP) [2]. The addition to RTMP is a simple encryption layer.

Encryption layer. RTMPE generates a stream key using a Diffie-Hellman [29] key exchange. Once this key is agreed upon, the entire communication stream is encrypted using RC4 [1]. No extra encryption is done on the media itself.

Performance. Any online analyzer running against RTMPE must be fast enough to allow the processing of the data stream without dropping the connection.

2.3.3 Spotify

Spotify implements a custom protection scheme to prevent duplication of their content. This scheme was reverse-engineered by the Despotify Project in their attempt to create an interoperable client [5]. The scheme uses a stream cipher to protect its communication, and, in addition, it encrypts each individual song.

Stream cipher. The Spotify client performs a key exchange with the server to create a key to be used for the remainder of the session. After the key is generated, the session is encrypted using a Shannon stream cipher [42].

Song encryption. Individual music files sent by Spotify in the encrypted stream are themselves encrypted with AES. The keys to this encryption are sent in the stream along with the music files. Upon receipt of a music file and its corresponding key, the Spotify client decrypts the file for playback. For offline playback, Spotify can cache this data.

Performance. An online analysis of Spotify must be fast enough to process the data stream without dropping the connection. Additionally, if the Spotify client runs too slow, it will mistakenly perceive that the connection to the server has been lost.

2.4 Bypassing DRM

As noted above, DRM methods tend to have unique workarounds, depending on their specific characteristics. For non-interactive multimedia, one general approach is called the Analog Hole [47]. The Analog Hole is a "flaw" in any DRM scheme, which is due to the fact that any media must eventually be consumed by a human. For example, a video will eventually have to be displayed on a screen and seen by someone's eyes. In the simplest setting, a human could just record the protected music or movie with a microphone or a camcorder. Programs [4] exist that will even record a movie as it is playing on the screen by scraping the screen's pixels.

However, since all the streaming media platforms known to us use lossy encoding for space and bandwidth-saving reasons, this type of DRM bypassing has the downside of a loss of quality due to the necessity to re-encode the captured audio and video. The only way to duplicate such content without quality loss is to capture the decrypted content after decryption but before decoding. There are two ways to do this: recovery of the keys used in the cryptographic process and the interception of the decrypted content. The former method requires approaches that may vary widely based on the DRM scheme and the type of encryption and key management used. Additionally, white-box cryptography [24] could be used to greatly complicate such an implementation by obscuring the usage of the cryptographic keys. The latter approach, which MovieStealer uses, allows us to intercept decrypted content irrespective

of the underlying encryption protocols. By doing this, it is possible to recover the original high-quality media sent by the media originator in a general way.

2.5 Cryptographic Function Identification

Since the identification of cryptographic functions is relevant to many other fields of study, and particularly relevant to malware analysis, other works have looked into identifying cryptographic routines.

An early approach to detecting decryption in memory is detailed by Noe Lutz [39]. This approach slows down the instrumented program by a factor of 2,400, an unacceptable slowdown for a high-performance media streaming application. Additionally, this approach detects encrypted data by measuring entropy. Such a detection would be unable to distinguish between encrypted and compressed (or, in our case, encoded) data.

Another approach, ReFormat [46], functions by detecting the flow of input data from a decryption routine to a handling routine followed by a flow of output data to an encryption routine. This approach does not work for our application domain for two reasons. Depending on the protocol, an analyzed media player might not necessarily encrypt a response. For example, the actual communication protocol of Microsoft's Silverlight streaming platform is not encrypted. As such, the client only decrypts the encrypted stream data, but does not have to encrypt any responses. Furthermore, ReFormat detects the transition from encrypted to decrypted data based on the percentage of arithmetic and bitwise functions processing it. However, since the decrypted stream in a media player is passed on to the decoding step, this heuristic does not necessarily hold true.

Dispatcher [20] is an approach that analyzes the data flow of bots to determine their communication protocol. To find the decrypted data, the system uses a similar method to ReFormat. Dispatcher also functions through offline analysis, and would be unsuitable for our application.

Another interesting approach is presented by Caballero, et al [21]. This approach is geared toward *removing* the decryption and decoding functionality in a malware program to easier interact with it (in the paper, the authors interacted with the malware to find bugs). This is not applicable to our case, since we gain no benefit from interacting with a media player directly. In any case, the overhead in this approach likely makes it unfeasible for use on large programs such as media players, although we are unable to verify this as the authors did not publish the tool itself.

BCR [19] is a tool that implements an algorithm, similar to MovieStealer's buffer detection, for detecting cryptographically-relevant loops and buffers. However, this approach has very heavy overhead, requires several similar executions, and relies heavily on offline analysis, which makes it ineffective for our target applications. Aside from the performance issue, this is also due to the fact that streaming media players are not completely deterministic because of changes in behavior due to network latency, user interaction, and other factors. These factors are often hard or impossible to control between executions, especially with complicated DRM platforms such as PlayReady. Additionally, by avoiding this requirement in MovieStealer, we are able to simplify our approach by not worrying about buffers being relocated by ASLR.

The approach described by Grobert, et al [32] also detects cryptographic primitives, but is another offline analysis and would not be performant enough for a media player. Additionally, this approach, along with other similar approaches that check for cryptographic primitives, would be sensitive to white-box obfuscation.

Finally, a recent result in this area of research is Aligot [22]. Aligot works by identifying loops in programs, identifying data flow between such loops, and comparing the result against reference implementations of cryptographic primitives. However, it also functions in an offline manner and (considering the amount of time its offline phase requires) would be too slow for a media player to function. Additionally, Aligot requires that the program being analyzed utilize a standard implementation of cryptographic primitives, while our approach avoids such an assumption.

These existing approaches are not adequate for breaking DRM in media players. Since the media services that we analyze have real-time requirements, any approach must have minimal overhead to function. However, these approaches were mostly designed to be run against small, non-demanding malicious programs. In general, they have high overhead and rely on offline analysis while MovieStealer is designed to be a fast, online approach. Furthermore, these approaches do not address the distinction between encrypted and encoded/compressed data with the regards to randomness as opposed to entropy, which is necessary to locate the appropriate buffer from which to extract the decrypted-but-encoded media stream.

3 Approach

Our intuition is that the authors of a media player would reuse existing, proven codecs for ease of deployment, performance, and reliability. Thus, at some point during the processing, one should see the data decrypted and sent to the media codec for decoding. By examining the data as it flows through the authorized media player, one can detect the point at which the player transforms the data from an encrypted stream to an encoded stream. Once this location in the program is detected, the decrypted stream can be dumped and reconstructed. Our approach leverages this observation and provides an automatic mechanism to break the DRM schemes of several popular streaming services.

The process of copying protected content can be divided

into three separate phases:

1. Analyze the way in which the authorized media player handles the encrypted stream and identify the point at which the stream is decrypted.
2. Dump this decrypted stream.
3. Reconstruct the original media file from the decrypted stream.

Normally, the first step would have to be done once per media player (or, depending on the DRM implementation, once per media codec), while the second and third steps would be repeated for each dumped movie.

Given an authorized media player executable, MovieStealer will execute the binary, trace its execution flow, monitor and log its data access, recover loops and buffers (defined as consecutive bytes of data), recognize the decryption step, dump the decrypted data, and construct a media file with the unprotected content.

3.1 Stream Decryption Analysis

The first step in the copying of protected content is the analysis of the authorized media player's processing of the encrypted stream. Of course, much of the code dealing with Digital Rights Management is heavily obfuscated, packed, or protected, and so our approach must be able to work with countermeasures such as dynamically generated functions. Therefore, the stream decryption detection is based on the dynamic analysis of the player application.

A media player processes a substantial amount of data in the course of downloading, decrypting, decoding, and playing media. Intuitively, such data, whether encrypted or decrypted, is stored in buffers in memory. While this data could conceivably be stored in evasive schemes (for example, splitting up buffers so that no two bytes are adjacent), we have not observed such evasiveness in the real-world applications that we have analyzed. Moreover, this would complicate the development process and would impede performance.

Thus, our goal in this step is to identify the location in the program where an encrypted buffer is turned into a decrypted buffer.

3.1.1 Loop Detection

The intuitive way to access data buffers is through a loop (or a loop-equivalent CPU instruction). As data decryption involves accessing the encrypted buffers, we would expect (and, indeed, this is what we have observed) it to be done using loops. Our intuition here is that a loop will exist that carries out a decryption operation on a small chunk of data. This loop (or, more precisely, its output) is what we are looking for. Hence, the first step of our solution is to automatically identify loops in the program.

Subsequent parts of our analysis work on loops rather than either functions or individual instructions for several reasons. First, loops are more likely to access a small number of buffers for a single purpose, while functions might access many buffers for several purposes. Secondly, a single instruction might only carry out a partial operation on the buffer. For example, a loop might carry out an entire decryption step while a single instruction in the loop might simply XOR two words together. Thus, by performing our analysis at the loop level, we can better see individual actions that a program carries out on its buffers. Thirdly, identifying functions within a program, without symbol tables and in the presence of obfuscation, is a complicated and error-prone task. We bypass this problem by operating on loops, which are more straightforward to identify. A loop can usually be identified as long as its basic blocks are executed at least twice.

```
        mov eax, 0
.head:
        mov ebx, (0x1000, eax, 4)
        mov (0x2000, eax, 4), ebx
        inc eax
        cmp eax, 5
        jne .head
```

Table 2: An example of a loop.

```
mov eax, 0xBAADF00D
xor dword ptr [esp], eax
```

Table 3: An example of an implicit read by a loop.

Although we perform our analysis on loops, our approach is inspired by some basic concepts taken from function analysis. A good example is the input and output of a loop. We mark all data that a given loop reads as its input, and all data that it writes as its output. Table 2 gives an example of a loop that reads 5 dwords from the buffer starting at 0x1000 as input and writes them to the buffer at 0x2000 as output.

It is important to note that, in the x86 architecture, data can be an input to a loop without being explicitly read by that loop. For example, Table 3 demonstrates such a case, where *esp*, despite not being explicitly read by the code in question, is an input to the XOR operation.

Our approach assumes that the decryption process happens inside a loop. More specifically, we expect to find a loop in the authorized media player that has at least one encrypted buffer as an input and at least one decrypted buffer as an output. We expect this decryption to be done in a loop because such DRM schemes on media must process large volumes of data, and the most efficient way of processing such data is through a loop or loop-like instruction.

Detecting the loops. Our approach to detecting loops is mainly inspired by LoopProf [40]. LoopProf maintains a Basic Block Stack (BBLStack) per thread. A BBLStack is a stack of basic block addresses. Whenever a basic block

is executed, its start address is pushed to this stack, and when the basic block exits, the start address is popped.

Our analysis routine is called every time a Basic Block (BBL) is executed. The analysis routine attempts to find the same BBL by tracing back in the BBLStack. If the same BBL is found in the BBLStack, the basic blocks between this BBL and the top of the stack are considered to be a loop.

Note that when using this approach, some additional care must be taken to avoid misdetection of recursive calls as loops.

Although the basic idea of loop detection is simple, much attention was given to performance. We explain our optimizations in detail in Section 4.2.

Maintaining a call stack. As described in LoopProf, loop detection by BBLStack can cause our program to identify loops that occur across function boundaries, which is often the case with recursive function calls. While this would not break our approach, we have chosen to detect and remove these loops to improve the performance of the analysis, given that we have not seen any example of decryption being done in a recursive fashion.

Using our call stack, we only check BBL inside the current frame when searching for loops. We maintain this call stack for every thread by instrumenting every call and return instruction. Of course, functions do not have to use these instructions, in which case one would still detect the blocks as a loop. In the cases we have examined cases, this is acceptable for our approach.

Apart from aiding in loop detection, the presence of a call stack allows us to identify loops that are used for multiple purposes. For example, one loop could be used both to encrypt and to decrypt buffers. In this case, if the loop is called by one function, it behaves like a decryption routine, and has a random input as well as a non-random output. However, when called by a different function, the loop might behave like an encryption routine, and would throw off our detection if we did not differentiate between these two cases. Table 4 illustrates this scenario. Differentiating between these two scenarios is important for our analysis, since we analyze all of the data read and written by each loop in aggregate across several runs. Thus, we must differentiate between the two execution paths of this loop in order to distinguish the two different cases. Therefore, a loop is identified not only by its basic blocks, but also by the top several functions on the call stack at the time it was called.

Detecting unrolled loops. Loops are frequently unrolled for increased performance. Specifically, the first or last few iterations are often unrolled, with the rolled loop present in case more data needs to be processed. In order to detect unrolled loops, we take note of the basic blocks that were executed between any two loops. We later check if these basic blocks do operations on the same buffer as either of the two loops.

```
void crypto_loop(void *key, void *in,
            void *out, int len);

void encrypt() {
  crypto_loop("key", decrypted,
              encrypted, len);
}

void decrypt() {
  crypto_loop("key", encrypted,
              decrypted, len);
}
```

Table 4: An example of both the encryption and decryption being done in one loop.

3.1.2 Buffer Identification

According to prior work in the field of data reverse-engineering, most buffers are accessed in loops [43]. Thus, having identified loops, we must then identify the buffers on which they operate. For the sake of performance, and unlike the approaches outlined in Howard [43] and REWARDS [37], which track the base pointers of and offsets into buffers by instrumenting every instruction, our approach is based on recording and analyzing reading and writing operations inside a loop. This is similar to what is implemented in BCR [19]. In addition, several heuristic methods are applied to improve the detection of the buffers. By applying these heuristics, even complex buffers such as the key permutation array used in RC4, which is accessed neither consecutively nor completely in most cases, can be identified by our approach.

Fetching memory access patterns. When MovieStealer is analyzing a loop, it dynamically instruments each read and write within that loop. For each such read and write, we record the target memory location that it accesses, the instruction pointer where the access occurs, and the size of the read or write. Note that some instructions, when called with specific operands, execute both a read and a write operation.

Whenever control flow leaves the loop, we move on to analyzing the loop's memory access patterns.

Analyzing memory access patterns. A loop can access a buffer in one of several different access patterns. Our approach focuses on detecting the following ones:

1. Consecutively accessing the buffer byte-by-byte.
2. Consecutively accessing the buffer dword-by-dword.
3. Consecutively accessing the buffer at single-byte offsets and reading a dword at a time.
4. Consecutively accessing the buffer using multimedia CPU extensions, such as SSE instructions [45].

Address	Step 1	Step 2	Step 3
0x1000	O (size 4)	C (element size 4)	C (element size 4)
0x1004	O (size 4)		
0x1008	O (size 4)	C (element size 4)	C (element size 4)
0x100c	O (size 4)		
0x1010	O (size 4)	O (size 4)	
0x1010	O (size 8)	O (size 8)	C (element size 8
0x1014			

Table 5: An example of the creation of composite buffers (C) from the memory read operations (original buffers O) of the code in Table 2.

5. Accessing the buffer in a predictable pattern. For example, two first bytes out of every three consecutive bytes are read in a buffer.
6. Accessing the buffer in an unpredictable pattern. In most such cases, the buffer is not fully accessed during the execution of a loop. For instance, accessing the key permutation array of the RC4 [1] algorithm.

To identify a read or write buffer, we perform our analysis in several steps. First, we classify each memory region affected by an individual memory access as an *original* buffer and sort them by their starting memory addresses. Then, we merge these buffers into *composite* buffers by recursively applying the following steps until there are no more candidates for merging. As we merge the buffers, we attempt to determine the size of the elements in each buffer.

- Two *original* buffers are merged if they are adjacent and are of equal size. In this case, the element size for the resulting composite buffer is set to the size of the two original buffers.
- Two *original* buffers are merged if they overlap, are of equal size, and their size is divisible by the size of the overlapping portion. In this case, the element size for the resulting compound buffer is set to the size of the overlapping region between the two original buffers.
- An *original* buffer is merged with a *composite* buffer if they are adjacent and the element size of the composite buffer is equal to the size of the original buffer.

This is applied recursively until there are no more *original* buffers that can be merged. At this point, any remaining *original* buffers are reclassified as *compound* buffers with an element size equal to their length. An example of this is detailed in Table 5.

This step merges the individual memory accesses into a preliminary representation of buffers. The sizes of these *composite* buffers will vary, but will be divisible by their element size. This representation is finalized in the next step, where the composite buffers are merged.

Merging composite buffers. Due to the way in which some buffers are accessed, they will be split into several composite buffers in the previous step. One example of this is the key permutation array used in RC4 [1]. This buffer usually has a size of 256 bytes, and is not likely to be completely read or written if there are less than 100 bytes to be decrypted. One approach is to aggregate the memory accesses over several different calls to the function to identify the buffer, but that brings up questions of when to terminate such an analysis. Therefore we use a simple heuristic to better identify such buffers: Given two existing composite buffers C and D, where buffer C starts at $addr_c$ and has a size of $size_c$, while buffer D starts at $addr_d$ and has a size of $size_d$, and $addr_d > addr_c$. We define the term gap ratio as the size of gap between buffers C and D divided by the sum of sizes of buffers C and D:

$$gratio(C,D) = \frac{addr_d - (addr_c + size_c)}{size_c + size_d}$$

We then perform the following algorithm:

1. If C and D have the same element size, and they are adjacent, they will be merged into a larger buffer.
2. If C and D are not adjacent, and they have the same element size, they will be merged if the gap ratio is less than 0.2. We determined this number experimentally. Of course, setting this threshold to a value too large will create false positives in the buffer detection (and will add noise to our subsequent statistical testing), while leaving it too small will cause us to miss parts of the buffers.

This algorithm is applied on the set of composite buffers until no more buffers can be merged.

Tracking unrolled loops. After the composite buffers are merged, we add any memory accesses done by blocks that are adjacent to the buffers and are identical to the blocks inside a loop. This allows us to catch the marginal parts of buffers that are modified by unrolled loops.

Data paths. After this step, we will have obtained a full list of buffers that are accessed inside each loop. We define a data path as an input-output buffer pair within a loop. A loop could have multiple data paths, as shown in Table 6. In the absence of detailed data-flow analysis, we conclude that every input buffer and every output buffer in a loop make a data path. Thus, in a loop with N input and M output buffers, we will have $N \times M$ data paths.

3.1.3 Decryption Detection

After identifying the buffers and the paths between them, the next step is to identify the buffer that holds the decrypted content. While a full analysis of every data path in a real-world application could be unfeasible due to the complexity of modern media players, we can utilize several heuristics to identify the data path that contains the decryption of the protected content. First of all, the data path that

```
        mov eax, 0
.head:
        inc eax
        mov ebx, (0x1000, eax, 4)
        mov (0x2000, eax, 4), ebx
        mov ebx, (0x3000, eax, 4)
        mov (0x4000, eax, 4), ebx
        cmp eax, 10
        jne .head
```

Table 6: An example of a loop with four data paths: 0x1000 to 0x2000, 0x1000 to 0x4000, 0x3000 to 0x2000, and 0x3000 to 0x4000.

	Input		Output	
Stage	E	R	E	R
Download	high	high	high	high
Decrypt	high	high	high	low
Decode	high	low	low	low

Table 7: The entropy (E) and randomness (R) of data paths when playing a protected media file.

we are looking for should have a similar throughput to the size of the media file. Additionally, since we are looking for a data path that has an encrypted input and a decrypted (but encoded with a media codec) output, we can utilize information theoretical properties of the buffers to improve our analysis.

We perform this step on the aggregated input and the aggregated output buffers of each data path. That is, we append all of the input and all of the output of a given data path across multiple executions of the loop in question, resulting in an overall input buffer and an overall output buffer. This allows us, for example, to analyze all of the output of a given operation across the runtime of the program. In the case of a decryption function, this will allow us to collect all of the decrypted content.

Entropy test. The data path in which we are interested will have an encrypted input buffer and a decrypted but encoded output buffer. The input buffer, being encrypted, will have very high entropy. The output buffer, being encoded (and effectively compressed), will *also* have very high entropy. We use this property to further filter out unrelated data paths.

This also helps filter out the decoding step. Media codecs are highly compressive functions, resulting in high-entropy buffers. On the contrary, a buffer of, for example, YUV color frames is likely to have a comparatively low entropy.

Randomness test. A fundamental property of encrypted data is that it is indistinguishable from random data. This is called *ciphertext indistinguishability*, and is a basic requirement for a secure cryptosystem [31]. Furthermore, randomness is very difficult to achieve, and is not a feature of data encoding algorithms. Such algorithms, which are essentially specialized compression algorithms, produce data with high entropy but low randomness. Thus, as shown in Table 7, we can distinguish between the encrypted and decrypted stream by using a randomness test.

The Chi-Square randomness test is one such test, designed to determine if a given input is random. Often used to test the randomness of psuedo-random number generators, we use it to determine whether or not the content of a buffer is encrypted. The implementation details of the Chi-Square randomness test is detailed by Donald Knuth [33] and its application to randomness testing is presented by L'Ecuyer, et al [36]. Our approach does not rely on the implementation details of the randomness test, and we have omitted them in the interest of space. Furthermore, the Chi-Square randomness test is not the only one that can be used; any measure of randomness of a buffer can be utilized for this purpose.

One important consideration is the amount of data that we should collect before performing our randomness test. A commonly accepted rule for the Chi-Square randomness test, mentioned by Knuth [33], is that given n, the number of observations, and p_s, the probability that n is observed to be in category s, the expected value $n \times p_s$ is greater than 5 for all categories s. We consider the contents of each buffer one byte at a time, giving us 256 categories of s. According to calculations presented by Knuth, we would need to collect 320 kilobytes of data for a reliable test. In fact, we carried out an empirical analysis of the minimum amount of data that needed to be analyzed to be confident of avoiding misdetection. The analysis determined that a safe threshold to avoid misclassifying random data as non-random is 800 kilobytes, and a safe threshold to avoid misclassifying non-random media data as random is 3800 bytes, both of which are easily feasible for any sort of media playback.

We have observed that the Chi-Square randomness test returns extremely low values (very close to 1.0) for encrypted data, and very high values (in the thousands) for encoded data.

3.2 Dumping the Stream

After the previous steps, we are able to identify the specific data path that has the encrypted input and the decrypted, but decoded, output. Then, our system instruments the authorized media player and dumps the output buffer.

3.3 Reconstructing the File

Finally, with the decrypted data available, the last step is to reconstruct the media file. In the trivial case, the DRM scheme works by encrypting the entire media file wholesale. This is simple to recover because the decrypted buffer

that we dump will then contain the whole, unprotected file. However, this is not the case in general. For example, the approach used in Microsoft's PlayReady DRM encrypts just the media stream, leaving the headers and metadata decrypted. Thus, the decrypted stream will contain the raw media stream, which cannot be directly played by a media player. In the general case, this is a problem of program analysis and writing an automated tool to reconstruct the file given an unknown protocol is quite complicated.

In order to recreate the media file in these circumstances, knowledge of the streaming/DRM protocol is required. For example, knowing that PlayReady encrypts the media stream, we wrote a reconstruction plugin to reconstruct the file with the newly decrypted media stream (and the already decrypted headers and metadata) so that it would be playable in conventional media players.

Depending on the protocol and the expertise of the operator, this stage can involve reading documentation, reverse engineering, and file analysis. However, at this point the automated decryption of the content, which is the central aim of our paper, is already completed. While the media content will need to be reconstructed for every dumped file, the development process is only required once per DRM platform (or, depending on the implementation, once per DRM platform/media codec combination).

4 Implementation

We implemented our system using the PIN [38] Dynamic Binary Instrumentation framework. We chose this tool for its ease of development, but our approach can be implemented on top of a full-system emulator such as QEMU in order to avoid anti-DBI techniques by the media-playing applications. However, the use of QEMU would raise the question of performance, since it is not clear if QEMU's dynamic recompilation of binaries can match the performance of PIN. Additionally, though our system is implemented under the x86 architecture, the approach is easily translatable to other architectures as well.

We will detail some of the specific implementation details related to the individual DRM platforms that we analyzed. Additionally, in the course of implementing our approach, we made several implementation decisions, which we will discuss hereinafter. After describing these, we will also detail optimizations that we used to increase the speed of our approach.

4.1 DRM Platforms

We specifically investigated three DRM platforms: Flash video with Amazon Instant Video and Hulu, Microsoft PlayReady with Netflix, and Spotify. Here we will give an overview of the protocols and the tools we developed to support them.

4.1.1 Flash RTMPE

Amazon Instant Video and Hulu both use the RTMPE protocol, developed by Adobe, to transmit video. RTMPE works by encrypting the whole media file on the fly before sending it across the network.

Since the entire file is encrypted, reconstructing it did not present a challenge because it was decrypted in a continuous manner in one function.

4.1.2 Microsoft PlayReady

Netflix uses Microsoft's Silverlight PlayReady DRM to protect its content. PlayReady presents several challenges.

Relocating code. In Silverlight, the actual routine used for decrypting AAC audio, WMV video and H.264 video is frequently relocated inside the process' memory space. We assume that this is done to frustrate would-be pirates. An additional benefit, given the required flexibility of the surrounding code, would be the ability to dynamically update the decryption routine over the network. However, we did not observe the latter ever occurring. Ironically, this evasive behavior gives us a clear signal that such code is *interesting*, and could enable us to prioritize it in our analysis.

To cover the case of relocated code, we identify such loops based on the non-relocating portions of their call stack and the hash of their basic blocks. This allows us to handle relocating code automatically as part of the normal analysis.

Disabling adaptive streaming. Netflix automatically adjusts the quality of the video stream to compensate for bandwidth and CPU inadequacies. This can result in a varied quality in the generated media file, which would lead to a confusing subsequent media consumption experience. Furthermore, because the MovieStealer implementation is extremely CPU-intensive, such adaptive streaming features will invariably select the stream with the worst available quality.

Our solution to this problem, specifically for Silverlight-based streaming services, is to use a Winsock introspection tool named Esperanza [7] to inspect the browser's traffic and filter the lower-bandwidth stream options out of the metadata. While this is a protocol-specific fix, a generalized version of this would be outside of the scope of this paper.

Partial encryption. PlayReady is hard to work with because it only encrypts the raw stream data of its media files. Header information and meta-data is not encrypted. Because of this, the decrypted file must be pieced back together by combining the original metadata and the dumped stream. Furthermore, some of the headers have to be modified to reflect the fact that the file is no longer encrypted.

4.1.3 Spotify

Spotify's distinguishing factor is the use of the Themida packer to frustrate our DBI platform. Since our instrumen-

tation is done dynamically, we would normally be able to copy protected content of packed programs. However, because Themida contains some evasive behavior that is able to confuse PIN, we had to use the OllyDBG debugger to first neutralize Themida's evasiveness by hiding PIN's presence. After this, we were able to extract music from Spotify, despite it being packed with the Themida packer. While this is not automated in *our* implementation, automating such anti-debugging practices is quite feasible.

Spotify encrypts its music files as a whole, so reconstructing them is straightforward.

4.2 Optimization

A basic implementation of our approach is able to detect and duplicate decrypted data in a program, but is not yet performant enough to analyze media players. To remedy this, we developed several optimizations. There are three stages that can be optimized: the selection of loops to analyze, the instrumentation and analysis of the loops themselves, and general performance optimizations.

Admittedly, some of the optimizations presented here are not automated. Specifically, limiting code coverage requires some domain knowledge to determine which code should be instrumented. However, MovieStealer can still function, albeit at a reduced speed, without this optimization.

4.2.1 Intelligent Loop Selection

Due to the overhead involved in instrumenting the memory operations of a loop and keeping track of the data that a loop is accessing, MovieStealer instruments a limited number of loops at a time. While a loop is instrumented, data from its buffers is saved and passed to our analyses. When a loop is determined by the randomness test to not be the decryption loop, or it is eliminated by one of the optimizations mentioned below, we discard the tracked state and instrument the next loop. To minimize the time necessary for MovieStealer to find the decryption routine, we need to select these loops in the most optimal matter.

Limit code coverage. Code coverage greatly influences the execution speed of MovieStealer. In most cases it is not necessary to instrument every module of the target process. For example, only the core libraries of Silverlight need to be instrumented to bypass PlayReady DRM, rather than the whole set of libraries of the browser. To reduce the number of loops that need to be analyzed, we only select the ones in the suspected DRM code, cutting out a significant amount of overhead.

On-demand instrumentation. Although we limit code coverage, there are still many instructions that are executed only once during initialization, which have nothing to do with decryption. Instrumenting and analyzing such loops would be a waste of resources. Inspired by PrivacyScope [50], we have designed MovieStealer to start after the program has initially loaded. After we load the authorized media player, we start MovieStealer and begin the media playback process. Thus, the initialization code will not be analyzed and MovieStealer will immediately begin zeroing in on the actual decryption functionality. We have observed that this significantly reduces the amount of loops that MovieStealer has to instrument and analyze.

Loop execution frequency. Additionally, we have observed that, in a streaming media player, the decryption routine is usually one of the most frequently-executed loops. This is because additional media is constantly being loaded over the network and must be constantly decrypted. On the other hand, loops pertaining to other functionality (for example, UI processing), are executed comparatively less frequently. To take advantage of this, we prioritize these loops for analysis ahead of less-frequently executed loops.

Static instruction analysis. As described in prior works [20, 46], code that carries out cryptographic functionality tends to utilize a large amount of certain types of operations. To optimize our analysis, we statically analyze the amount of arithmetic and bitwise operations in every loop and de-prioritize loops that lack such operations.

Additionally, we have observed that decryption routines often contain unrolled loops for increased performance. As such, we assign a higher priority to loops that are unrolled. We statically detect unrolled loops by detecting a repeating pattern of instructions before or after a loop body. While this is a very simplistic approach to unrolling detection, we feel that it is adequate. It works for the code that we have observed in our analyses, and if it fails to detect an unrolled loop, such a loop would still be analyzed later.

Loop hashing. In order to allow MovieStealer to function over several executions of a program, we save the results of our analyses for analyzed loops. We identify loops using a tuple consisting of the offsets of the basic blocks comprising the loop from the base address of their module, and the name of the module. When the analysis of a loop is finished, the results are saved before the state for the loop is discarded. This allows us to keep results over multiple executions of MovieStealer in case it takes an exceptionally long time to identify the decryption point. While this optimization can be useful, we did not run into any cases where we had to rely on it.

The astute reader will note that the relocating loops of DRM schemes such as Microsoft PlayReady will not be successfully recorded by this approach. However, this optimization would still allow us to avoid reanalyzing the majority of loops in a program, and being able to thus focus on just the relocating ones will greatly reduce the time required for MovieStealer to identify the decryption loop.

4.2.2 Improved Instrumentation

Intelligently selecting loops to instrument greatly improves MovieStealer's performance, but lots of time is

Stage	Input bandwidth	Output bandwidth
Download	roughly S	roughly S
Decrypt	roughly S	roughly S
Decode	roughly S	greater than S

Table 8: The bandwidth of data paths when playing a protected media file of size S.

still spent analyzing loops that turn out to be unrelated to decryption. For loops that handle a lot of data, this data needs to be analyzed in a performant fashion. However, when instrumenting loops that do not handle much data, much time is spent waiting to acquire enough data for the statistical tests. To further optimize this, we created several approaches to increase the performance of loop instrumentations and to decrease the time necessary to arrive at a classification.

Bandwidth filtering. Since protected media needs to be decrypted before being played, we should be able to find the decryption loop more efficiently by examining its data throughput. We define the input bandwidth of a data path as the amount of data in the aggregated input buffer and the output bandwidth of a data path as the amount of data in the aggregated output buffer. In Table 8, we detail the steps that an authorized media player takes when playing protected content, along with the expected input and output bandwidth of these functions. Intuitively, a loop that is decrypting the network traffic should have a similar bandwidth to the network traffic itself.

We carry out a bandwidth check on each instrumented loop every two seconds and compare it against the network traffic (for streaming media players) or the disk traffic (for GPG). Empirically, we determined that it's safe to discard a loop after 20 seconds if it fails the bandwidth test at least 60% of the time. A loop is considered to have failed a bandwidth test if its bandwidth is not within 60% of the expected bandwidth.

Avoiding unnecessary data copying. For the randomness test, the entropy test and the data dumping, we must record data chunks that are read or written during the loop execution, as described before. Since memory operations happen very frequently, performance is critical in tracking these reads and writes. Our approach must fulfill these basic requirements:

1. Moving, copying and modifying data as little as possible.
2. Imposing as little overhead as possible for addressing the buffer.

We did not include thread safety as one of the basic requirements, as in real-world media players few buffers are accessed simultaneously by multiple threads. We assume that programs that do access buffers concurrently will handle their own synchronization.

We have different strategies for reading and writing. For written data, rather than logging what is written, two variables holding the starting address and the ending address are maintained for every buffer. Each time a buffer write occurs, we update the starting address and ending address so that they correctly reflect the start and end positions that are written. As we expect these buffers to be consecutive, there is no problem with expanding the margins over bytes that are not read yet. For the randomness and entropy tests, MovieStealer analyzes every byte in the buffers between the start and end positions.

For content that is read out of buffers, we have a different strategy. As data being read during a loop might be overwritten inside the same loop, our write-buffer strategy does not always work. Hence it is necessary for MovieStealer to not only record the memory ranges, but also record the data located at the memory ranges at the time that reading happens. It is important to note that memory reading is not always consecutive nor always starts from the beginning of the buffer. Thus, through the single run of a loop, only parts of a buffer might be updated. To achieve better performance, we try to avoid re-copying unchanged data. This is done by treating each buffer as a concatenation of 4,096-byte blocks. As a loop executes, we mark the blocks that it modifies, and copy only the modified blocks when it exits. Our copied-off buffer is an array of pointers to these blocks. Any unchanged blocks on a new run are stored as pointers to previously-copied versions of that data.

4.2.3 Other Optimizations

Call stack optimization. To improve performance, a *call stack key* is maintained for each thread, and is updated each time a call or ret instruction is executed. When a new function is called, its start address is XORed onto the call stack key when the function is added to the call stack. When the program is about to return from a function, we pop the function from the call stack and wipe it from the call stack key by XORing its start address again. This way, we can use the call stack key instead of the whole call stack to identify a given loop. A dword comparison has considerably less overhead than a list comparison and, in practice, we have not seen any call stack key collisions due to this in our experiments.

This optimization is especially useful in loop selection, loop analysis, and data dumping.

5 Evaluation

In the course of our evaluation, we strived to demonstrate two things: that our optimizations work and are effective at improving performance, and that MovieStealer is an effective tool for bypassing the DRM of streaming media services. Since most of the streaming media services do not function at all without our optimizations, we

ran the optimization evaluation on GPG, an open source cryptography suite. GPG has fewer real-time processing requirements than real-world media players and as such works despite high overhead from unoptimized analyses.

We evaluated MovieStealer's effectiveness on a series of online streaming services, including Netflix, Hulu, Amazon Video, and Spotify. Our experiments consisted of loading the streaming application (in all cases except for Spotify, this was done by visiting the appropriate webpage in the browser. Spotify is a stand-alone application), starting MovieStealer, and playing a video or a song. MovieStealer would then pinpoint the decryption location and, on future runs, would begin dumping the media file. The reconstructor would then be run to create a playable media file. We verified that the media file was playable by playing it in a different, unauthorized player.

We carried out three experiments for each DRM platform, treating Hulu and Amazon Video as a single platform. For each experiment, we started MovieStealer from scratch. We recorded the number of loops identified, the loops analyzed before MovieStealer zeroed in on the sensitive loop, the total amount of analyzed loops that contain detected buffers, the total number of buffers identified, the total number of decryption loops that MovieStealer identified, and the total time until data could start being dumped. In all of the experiments, the loop responsible for decrypting the encrypted content was partially unrolled as a performance optimization.

To the best of our knowledge, MovieStealer is the first publicly described approach with the ability to successfully copy content protected by Microsoft PlayReady DRM without screen scraping techniques, as well as the first implementation to do cryptographic identification and copying of content at runtime.

MovieStealer was able to function on all DRM approaches that we evaluated.

Effect of optimizations. We carried our our optimization evaluations by executing MovieStealer against GPG as it decrypted a video file. First, we measured the performance of MovieStealer with all optimizations enabled, then measured the performance of first the callstack key optimization and then the code coverage limit optimization by running MovieStealer with all other optimizations enabled, and finally enabled some of our optimizations one-by-one to demonstrate their effects. The results can be seen in Table 9.

Necessary optimizations. Some of our optimizations were necessary to get the media players to function at all. As described in Section 2.3, these media players are high-performance pieces of software with some real-time requirements. For example, Netflix implements content expiration and has minimum performance requirements below which it will not play videos, and an unoptimized approach fails to meet such requirements. We have found that it is possible to analyze the streaming media players by enabling, at minimum, all of the loop selection optimizations.

Optimizations enabled	LT	S
All	7	31
All but callstack key	6	47
All but limit code coverage	10	34
Only limit code coverage	9	65
Only static instruction analysis	10	49
Only bandwidth filtering	35	180
Only execution frequency	40	3,480

Table 9: Results for GPG. LT = loops traced, S = total seconds before the decryption loop was identified.

Experiment no.	1	2	3
Loops identified	1,529	1,258	1,647
Buffers identified	14	6	1
Loops traced	46	35	62
Seconds elapsed	281	146	175

Table 10: Results for Amazon Video and Hulu

Experiment no.	1	2	3
Loops identified	2,876	2,274	2,950
Buffers identified	88	80	152
Loops traced	8	58	54
Seconds elapsed	86	110	191

Table 11: Results for Netflix

Experiment no.	1	2	3
Loops identified	2,305	1,845	1,667
Buffers identified	60	69	63
Loops traced	224	204	138
Seconds elapsed	536	739	578

Table 12: Results for Spotify

Non-determinism. Non-determinism is introduced into the results from several sources. To begin with, the programs in question are complex and multi-threaded, and rely on external resources to function. This means that the sequence that code is executed (and that MovieStealer analyzes it) in varies between runs.

Additionally, MovieStealer starts on demand, so it might begin analyzing different parts of the program in different runs. This will also make it analyze code in different order. Finally, the code relocations used by PlayReady DRM adds extra indeterminism to the mix. However, this does not have an effect on the final, successful decryption result.

6 Discussion

The expected use of an approach such as MovieStealer would be to save streamed movies either for later watching or for sharing with others. The latter approach is, of course, illegal. Our intention is not to aid illegal activity, and we discuss this further in Section 7.

It is also important to stress that in order for MovieStealer to function, the user must be authorized to play the content in the first place.

One possibility for future direction is a look into automatically cracking HDCP-protected content. Since the master keys are leaked, it might be possible to analyze encrypted-encrypted data paths and attempt to automatically use the HDCP keys to decrypt the content further. With the relatively low amount of buffers identified in the video experiments, this might be feasible from a performance standpoint. This would allow MovieStealer to function on devices with dedicated hardware for hiding content as it's re-encrypted for HDCP.

Another potential direction would be use MovieStealer to automatically recover encryption keys from running software. After detecting a decryption loop, MovieStealer could check the other inputs to that loop or to other loops that touched the encrypted buffer to determine if such inputs are the keys to the encryption.

Furthermore, it would be interesting to investigate the use of our approach to inform systems such as Inspector Gadget [34] in order to automatically export the encryption/decryption functionality of programs.

6.1 Countermeasures

Although our approach proved to be effective on current online streaming services, there are steps that could be taken by the authors of the DRM schemes to protect themselves against MovieStealer.

Anti-debugging. Applying extreme anti-debugging and anti-DBI techniques would prevent our implementation, *in its current form*, from working. However, nothing prevents one from implementing MovieStealer in a full-system emulator such as QEMU [16], rendering the approach immune to such evasions.

Attacking our loop detection. There are several ways to prevent MovieStealer from properly detecting loops within a program. A full unrolling of relative loops could effectively prevent the real loop from being detected by MovieStealer. However, full unrolling will result in loss of flexibility of the loop, and detection might still be possible using pattern matching approaches. Alternatively, protecting sensitive program modules by using virtual machine interpreted instructions would be very effective, as most of our loop identification approaches would not work. However, the performance penalty for doing this would likely be unacceptable.

Attacking the buffer detection. We cannot properly analyze a buffer that has a nonconsecutive layout in memory. For example, if a buffer only occupies one byte every three bytes, these bytes will not be identified as a valid byte array, let alone a buffer. We have not seen these techniques being used, and implementing them will likely carry an overhead cost. However, it is a definite possibility with modern hardware.

Along these lines, an effective countermeasure would be a functional hardware DRM scheme. However, it is not clear how to implement this in a way flexible enough to be resistant to events such as key leaks while being secure enough to be resistant to bypass.

Attacking the decryption detection. One very effective countermeasure would be to intersperse non-random data in the encrypted buffers, and to insert random data into the decrypted buffer. This would lower the randomness of the encrypted buffer and raise the randomness of the decrypted buffer, possibly defeating our analysis. The decoder would then be modified to ignore the inserted random bytes so that it can successfully replay the video. It is important to note that this approach would require a modification of the decoder, as removing the random bytes beforehand (and reducing the randomness of the buffer in question) would trigger MovieStealer's decryption detection.

Attacking the pirates. Watermarking has proven to be incredibly effective in tracking piracy. The originator could watermark the media [28, 17, 18], and in the event of piracy, the pirates could be identified by this watermark. This is a very effective technique, and it has been used to successfully track down pirates [13, 6]. While some research has been done toward the circumvention of watermarks [26, 35], a watermark-related arms race might be easier for content providers than the design of mechanisms to counteract approaches similar to MovieStealer.

7 Ethical and Legal Issues

In this section, we discuss the ethical and legal implications of our work.

First of all, obviously our work was never motivated by the desire to obtain protection-free copies of the media for re-distribution (piracy) or to create and distribute tools that would allow others to bypass content protection mechanisms.

Our goal was to analyze the security of the cryptographic mechanisms used by these emerging services, and to develop an approach that would demonstrate the general fallacy behind these schemes, in the hope that our findings would prompt the development of new, more secure approaches to content protection that are not vulnerable to our attack. This is especially important if cryptography-based protection mechanisms are touted to "protect" user-generated content (e.g., independent movies distributed

exclusively through streamed media) and give to the content authors (i.e., the users of the distribution service) a false sense of security with regards to the possibility of malicious third parties stealing their content.

The legality of this research is tightly related to the location where the research is performed. For example, there are some subtle but important differences between the laws in the United States and the laws of the European Union and Italy [23].

The research was carried out in the United States, and hence, it falls under the Digital Millenium Copyright Act [25]. The DMCA prohibits the circumvention of content protection mechanisms, but includes explicitly protection of security research (referred to as "Encryption Research" – see Section 1201(g) of the DMCA.) We feel that this research falls under this protection and is therefore legal. Citing from the DMCA document: "Factors in determining exemption: In determining whether a person qualifies for the exemption under paragraph (2), the factors to be considered shall include the information derived from the encryption research was disseminated, and if so, whether it was disseminated in a manner reasonably calculated to advance the state of knowledge or development of encryption technology, versus whether it was disseminated in a manner that facilitates infringement under this title or a violation of applicable law other than this section, including a violation of privacy or breach of security."

We feel that the way in which this research is disseminated is clearly focused on advancing research and not to facilitate infringement. In fact, we have chosen not to publicly distribute the source code of our tool or to provide ways to easily attack specific technologies. In addition, with the help of the Electronic Frontier Foundation, we contacted each of the companies involved in order to disclose these DRM workarounds responsibly. Microsoft was notified because they are the vendor of the Silverlight DRM used in Netflix. Adobe was notified because they are the vendor the RTME implementation for Amazon and Hulu. Netflix, Amazon, and Hulu were notified because the DRM being bypassed is used by their services. Spotify was in the unique position of falling into both categories. Of course, we contacted them as well.

Of the companies contacted, Netflix, Amazon, and Hulu did not respond to our initial or follow-up contacts, nor when contacted through EFF's channels of communication. However, Microsoft, Adobe, and Spotify responded, acknowledged the issues, and discussed workarounds. All three companies reviewed our work, provided comments for this paper, and encouraged its publication, for which we are grateful.

In summary, our goal is to improve the state-of-the-art in cryptographic protection and not to create and distribute tools for the violation of copyright laws.

8 Conclusions

In this paper, we have proposed MovieStealer, a novel approach to automated DRM removal from streaming media by taking advantage of the need to decrypt content before playing. Additionally, we have outlined optimizations to make such DRM removal feasible to do in real-time, and have demonstrated its effectiveness against four streaming media services utilizing three different DRM schemes.

Acknowledgements We would like to thank representatives from Microsoft, Spotify, and Adobe for their feedback in regards to the drafts that we sent them. Additionally, we are eternally grateful to the EFF and UCSB's legal counsel for their help with legal and ethical concerns during the publication process. Finally, we thank Dr. Jianwei Zhuge for his advice.

This work was supported by the Office of Naval Research (ONR) under Grant N00014-12-1-0165 and under grant N00014-09-1-1042, and the National Science Foundation (NSF) under grants CNS-0845559 and CNS 0905537, and by Secure Business Austria. This work was partly supported by Project 61003127 supported by NSFC.

References

[1] RC4, 1994. http://web.archive.org/web/20080404222417/http://cypherpunks.venona.com/date/1994/09/msg00304.html.

[2] RTMP, 2009. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf.

[3] Adobe RTMPE, 2012. http://lkcl.net/rtmp/RTMPE.txt.

[4] Audials software, 2012. http://audials.com/en/how_to_record_stream_capture_music_videos_movies_from/netflix.html.

[5] The despotify project, 2012. http://despotify.se/.

[6] E-city nabs pirates using thomson watermarking tech, 2012. http://businessofcinema.com/bollywood-news/ecity-nabs\%2Dpirates-using-thomson\%2Dwatermarking\%2Dtech/27167.

[7] Esperanza project, 2012. http://code.google.com/p/esperanza.

[8] Freakonomics - How Much Do Music And Movie Piracy Really Hurt the U.S. Economy?, 2012. http://www.freakonomics.com/2012/01/12/how-much\%2Ddo-music\%2Dand-movie\%2Dpiracy\%2Dreally-hurt\%2Dthe-u-s\%2Deconomy/.

[9] High-bandwidth Digital Content Protection System - Interface Independent Adaptation - 2.2, 2012. http://www.digital-cp.com/files/static_page_files/6FEA6756-1A4B\%2DB294\%2DD0494084C37A637F/HDCP\%20Interface\%20Independent\%20Adaptation\%20Specification\%20Rev2_2_FINAL.pdf.

[10] Microsoft PlayReady DRM, 2012. http://msdn.microsoft.com/en-us/library/cc838192(VS.95).aspx.

[11] Microsoft protected media path, 2012. http://scholar.google.com/scholar?hl=en&q=protected+media+path&btnG=&as_sdt=1%2C5&as_sdtp=.

[12] Nation of unrepentant pirates costs $900m, 2012. http://www.smh.com.au/technology/technology-news/nation-of\%2Dunrepentant-pirates-costs\%2D900m-20110305\%2D1bix5.html.

[13] Porn studio awarded 1.5 million from man who shared 10 movies, 2012. http://www.slate.com/blogs/trending/2012/11/02/kywan_fisher_flava_works_wins_1_5_million_in_biggest_ever_torrent_judgement.html.

[14] SOPA: How much does online piracy really cost the economy?, 2012. http://www.washingtonpost.com/blogs/ezra-klein/post/how-much\%2Ddoes-online-piracy\%2Dreally-cost-the\%2Deconomy/2012/01/05/gIQAXknNdP_blog.html.

[15] Spotify DRM, 2012. http://www.defectivebydesign.org/spotify.

[16] F. Bellard. QEMU, a fast and portable dynamic translator. USENIX, 2005.

[17] J. Bloom and C. Polyzois. Watermarking to track motion picture theft. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 1, pages 363–367. IEEE, 2004.

[18] L. Boney, A. Tewfik, and K. Hamdy. Digital watermarks for audio signals. In *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on*, pages 473–480. IEEE, 1996.

[19] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

[20] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634. ACM, 2009.

[21] J. Caballero, P. Poosankam, S. McCamant, D. Song, et al. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 413–425. ACM, 2010.

[22] J. Calvet, J. M. Fernandez, and J.-Y. Marion. Aligot: Cryptographic function identification in obfuscated binary programs. 2012.

[23] R. Caso. *Digital Rights Management: Il commercio delle informazioni digitali tra contratto e diritto d'autore*. CEDAM, 2006.

[24] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. A white-box des implementation for drm applications. In *Digital Rights Management*, pages 1–15. Springer, 2003.

[25] U. S. Congress. Digital Millennium Copyright Act, October 1998.

[26] I. Cox and J. Linnartz. Some general methods for tampering with watermarks. *Selected Areas in Communications, IEEE Journal on*, 16(4):587–593, 1998.

[27] G. Danby. Key issues for the new parliament 2010 - copyright and piracy, 2010. http://www.parliament.uk/documents/commons/lib/research/key_issues/Key%20Issues%20Copyright%20and%20piracy.pdf.

[28] E. Diehl and T. Furon. © watermark: Closing the analog hole. In *Consumer Electronics, 2003. ICCE. 2003 IEEE International Conference on*, pages 52–53. IEEE, 2003.

[29] W. Diffie and M. Hellman. New directions in cryptography. *Information Theory, IEEE Transactions on*, 22(6):644–654, 1976.

[30] Engadget. HDCP 'master key' supposedly released, unlocks HDTV copy protection permanently, 2010. http://www.engadget.com/2010/09/14/hdcp-master\%2Dkey-supposedly\%2Dreleased-unlocks\%2Dhdtv-copy\%2Dprotect/.

[31] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*, volume 2. Cambridge university press, 2004.

[32] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In *Recent Advances in Intrusion Detection*, pages 41–60. Springer, 2011.

[33] D. Knuth. *The art of computer programming*. addison-Wesley, 2006.

[34] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 29–44. IEEE, 2010.

[35] G. Langelaar, R. Lagendijk, and J. Biemond. Removing spatial spread spectrum watermarks. In *Proceedings of the European Signal Processing Conference (EUSIPCO98), Rodes, Greece*, 1998.

[36] P. L'Ecuyer. Testing random number generators. In *Winter Simulation Conference: Proceedings of the 24 th conference on Winter simulation*, volume 13, pages 305–313, 1992.

[37] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. 2010.

[38] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[39] N. Lutz. Towards revealing attackers intent by automatically decrypting network traffic. *Master's thesis, ETH Zuerich*, 2008.

[40] T. Moseley, D. Grunwald, D. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. LoopProf: Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[41] M. Peitz and P. Waelbroeck. Piracy of digital products: A critical review of the economics literature. 2003.

[42] C. Shannon. Communication theory of secrecy systems. *Bell system technical journal*, 28(4):656–715, 1949.

[43] A. Slowinska, T. Stancescu, and H. Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of NDSS*, 2011.

[44] F. A. Stevenson. Cryptanalysis of contents scrambling system, 2000. http://web.archive.org/web/20000302000206/www.dvd-copy.com/news/cryptanalysis_of_contents_scrambling_system.htm.

[45] S. Thakkur and T. Huff. Internet streaming SIMD extensions. *Computer*, 32(12):26–34, 1999.

[46] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. *Computer Security–ESORICS 2009*, pages 200–215, 2009.

[47] Wikipedia. Analog Hole - Wikipedia, the free encyclopedia, 2012. [Online; accessed 09-Nov-2012].

[48] Wikipedia. DeCSS - Wikipedia, the free encyclopedia, 2012. [Online; accessed 09-Nov-2012].

[49] Wikipedia. Software protection dongle - Wikipedia, the free encyclopedia, 2012. [Online; accessed 09-Nov-2012].

[50] Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical report, Tech. Rep. EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.