Content protection for games

G. Myles
S. Nusser

In this paper we review the state of the art in content protection for video games by describing the capabilities and shortcomings of currently deployed solutions. In an attempt to address some of the open issues, we present two novel approaches. The first approach uses branch-based software watermarking to discourage and detect piracy through a registration-based system. In the second approach, based on the parallels between games and premium audio and video content, we propose the use of current physical-media copy-protection technologies for gaming content. In particular, we focus on broadcast encryption technology. The use of an open, standard-based architecture enables the development of a more restrictive protection system for games. Finally, we demonstrate how the proposed protection mechanisms can be applied to video-game copy protection through five scenarios.

INTRODUCTION

Before the extensive availability of the high-speed Internet, the distribution of pirated software involved the transfer of a physical copy, that is, the transfer of a storage device such as a diskette or a compact disk. This limited the rate at which illegally copied games could be distributed, and thus the associated losses could be absorbed. However, the recent advances in technology, the ease of access to the Internet, and the widespread use of peer-to-peer applications made the physical copy obsolete. Software piracy is now a widespread, decentralized problem in which millions of people take part. The protection afforded by the legal system is no longer easy to enforce or cost effective. Thus, the gaming industry now relies on technological mechanisms to deter, detect, and prevent piracy.

In 2004, when sales of video games in the United States set a record at $7.3 billion,[1] the industry lost more than $1.8 billion to global piracy.[2] Unfortunately, the lost revenue is due to a variety of different piracy-related attacks, such as illegal copying, counterfeiting, and distributing. The ramifications associated with piracy propagate throughout the gaming community. The obvious victims are the game-producing companies, but the more peripheral victims are often not recognized. Hardware producers rely on video game players to buy the newest, fastest, and most expensive products to play increasingly resource-intensive new games. For example, in September of 2003 a significant portion of the source code of the game *Half-Life*** *2* was stolen. When the theft, which occurred before the release of the game, was discovered, the game maker delayed its release in

order to develop a new and different version of the game. This delay led to potential revenue losses for ATI, a graphics card company. As a marketing strategy to entice consumers to upgrade, they planned to distribute free versions of the game with their latest graphics. Due to the delayed release, the game was not available in time. The video game industry must also contend with unscrupulous retailers who are able to significantly increase their profits by producing and selling illegal copies.

To develop techniques for the protection of gaming content it is necessary to understand the mechanisms used by attackers to bypass copy-protection technologies. The attack mechanisms in use today target both the executable software and the console devices on which many games are played. In this paper we are concerned with the following types of attacks:

1. Sharing of installation media (e.g., illegal sharing of an installation CD from a legally purchased game package).
2. Creating and distributing illegal copies of installation media.
3. Disabling or bypassing copy-protection technology so that the game can be illegally redistributed (e.g., disabling the license check by modifying the game code).
4. Execution of illegally copied games on a traditional PC.
5. Execution of illegally copied games on a console gaming system.
6. Execution of illegally copied games on a gaming system that has been modified to run games using alternate media formats such as a standard CD.

It is important to note that the third type of attack, in which the game itself is modified, is distinct from the last one, in which the gaming system is modified. Modification of the gaming system to bypass copy protection is commonly referred to as *modding*.

In this paper we present two novel approaches for game protection. The first approach is based on a particular software watermarking technique, the branch-based software-watermarking algorithm, developed by the authors at the IBM Almaden Research Center.[3] Software watermaking is one of many techniques currently being studied to prevent or discourage software piracy and copyright infringement. The basic idea of watermarking is to embed a unique identifier in the program. Depending on the identifier, it can be used to indicate the author or the legal purchaser of the program. By incorporating ideas from code obfuscation (to aid in preventing reverse engineering) and software tamper detection (to thwart attacks such as the application of semantics-preserving transformations), the technique makes it possible to detect theft through the implementation of a registration-based system. The algorithm involves redirecting branch instructions to a special function known as a *branch function*. This function is responsible for computing the program's "fingerprint" and controlling the execution. The branch-based software-watermarking algorithm makes several improvements over previously proposed watermarking techniques:

- The technique simultaneously provides proof of authorship and the ability to trace the source of the illegal distribution.
- The technique demonstrates a significantly higher level of resilience to attack without significant overhead.
- The technique provides a means for distributing prepackaged, fingerprinted software whose link to the consumer is established at registration time.

In the second approach we show how games are similar to premium audio and video content in their need for copy protection. Based on these parallels, we propose the use of current copy-protection technologies for physical media in the battle against video game piracy. In particular, we focus on the broadcast encryption technology developed by 4C Entity, LLC, a consortium founded in 1998 by IBM, Matsushita Electric Industrial Corporation, Toshiba Corporation, and Intel Corporation to implement CPRM (Content Protection for Recordable Media). In this scheme, the game executable is stored in an encrypted state on a disk and depends on the presence of protected media (such as an SD memory card) to execute. As a consequence, multiple copies of a game can be created by an end user, but they cannot be executed concurrently. This is a more user-friendly approach than a per-PC license, which does not allow an end user to install a game on a second machine.

Content protection in the gaming industry is obviously an area of intense interest for developers

and producers of games. It is also of interest to game device makers, who often sell the device at a loss and instead draw their profit from royalties on software sales. Unfortunately, the gaming industry currently makes use of proprietary methods in the development of copy-protection technologies. Whereas this approach is generally effective against the occasional copier, such protection is usually vulnerable to an attack by an experienced hacker. Our approaches are based on open standards: the branch-based watermarking algorithm is publicly available, and the techniques for protection of physical-media content are based on open standards. Consequently, the techniques we discuss here are likely to be scrutinized for pitfalls by a large community, are based on a strong technical basis, and are likely to lead to more robust protection mechanisms for games.

The rest of the paper is structured as follows. In the section "Gaming and piracy," we present an overview of the state of the art in copy-protection technologies for PC and console-based gaming. Then, in the section "Using software watermarking to combat game piracy," we discuss our branch-based software-watermarking algorithm and the ways it can be used to prevent game piracy. In "Content protection for physical media," we briefly review the evolution of copy protection for audio/video content and for recordable media, with a particular focus on broadcast encryption, a cryptographic-key management technology. Finally, through five deployment scenarios described in the section "Content protection for games: Deployment scenarios," we demonstrate the viability of the proposed techniques to prevent common game piracy attacks. We thus demonstrate that current copy-protection techniques that were developed either for software or for video and audio content can be successfully applied to combat piracy in video games. In the "Conclusion" section, we summarize our main results.

Throughout the paper we use a variety of terms which could have multiple meanings. This paragraph clarifies our usage. We use the term *video games* (or games, for short) to refer to games that are played either on a PC (PC-based game) or specialized hardware known as a video game console (console-based game). The techniques we discuss here apply both to prepackaged games and games that are downloaded from the Internet. Whereas *copy protection* refers to preventing copying of copyrighted material, *content protection* is more general and includes other violations of intellectual property, such as performing illegal modifications to proprietary software. We use the term *premium audio and video content* interchangeably with *premium entertainment content.* We use the term *optical media* to refer to media such as the compact disk, for which the reading or writing of information is performed through optical techniques. We use the term *hackers* to refer to people involved in software piracy.

GAMING AND PIRACY

To combat the high level of piracy, the video game industry has taken a variety of actions that include both deterrent and preventative techniques. In this section we present the current state of the art in both hardware- and software-based protection mechanisms.

Hardware-based protection techniques

A variety of hardware-based techniques have been used in video game anti-piracy measures. These techniques are typically able to provide a higher level of protection than their software-based counterparts. However, the techniques are generally more expensive to produce and often cumbersome for the end user.

One of the first hardware-based, anti-piracy techniques deployed in the game industry was the *dongle*.[4] The dongle, a hardware device commonly distributed with a piece of software to prevent unauthorized execution, is typically connected to an I/O port, such as a serial or parallel port. As the software executes, it periodically queries the dongle, which returns the output of a secret function. If communication fails or the result of the query is incorrect, the software will eventually produce incorrect results or fail entirely.

The use of a dongle as a protection mechanism has several drawbacks, the first of which is the cost. The cost of the dongle, approximately $10 in the United States, further increases the cost of the game. Second, the use of a dongle limits the distribution options. In particular, when a game is sold and distributed over the Internet, the inclusion of a dongle is not feasible. Finally, code for dongles is often "cracked" shortly after release. This is generally accomplished by "disassembling" the

game code, identifying the calls to the dongle, and then bypassing those calls. After the dongle is cracked, a code patch is usually distributed so that anyone can play the game without the required dongle. Such an instance occurred with the *Robocop*** *3* game for the Amiga** platform. The antipiracy dongle had to be connected to one of the joystick ports for the game to run. A few days after its release in April of 1992, the dongle was cracked.

A second hardware-based piracy prevention technique is *tamperproof hardware*.[5] Tamperproof hardware is a way to secure parts of the hardware, such as the use of a computer chip, from being observed by a hacker. By executing the software in a secure context the pirate is unable to gain access to the application code and identify the code to be bypassed. This piracy prevention technique is feasible for console-based systems but has limitations for PC games. Because a user must purchase a console to even play a game, the game developers can make use of this technology, but the additional cost of requiring all PC game users to have tamperproof hardware is not currently a viable solution.

The development of the Trusted Platform Module[6] (TPM) is one example of using tamperproof hardware to prevent software tampering. The TPM is a special chip developed to enable trusted computing features. The four essential features include:

1. *Secure I/O*—Input and output are verified by performing a checksum of the software used for I/O.
2. *Memory curtaining*—The hardware prevents a program, including the operating system, from reading or writing memory used by another program.
3. *Sealed storage*—Information is protected by encrypting it with a key derived from the hardware or software currently being used.
4. *Remote attestation*—Changes to the computer are detected by having the hardware generate a certificate stating what software is in use. The certificate is presented to the remote party, generally through the use of public key cryptography, to demonstrate that the system has not been altered.

The TPM features have been incorporated into chipsets by Intel Corporation, Advanced Micro Devices, Inc., and IBM.

One mechanism used by hackers to bypass copy protection is the *mod chip*. This is a special chip added to the game console that is capable of modifying or disabling security mechanisms. Through the use of the mod chip a user can play games from other regions (installation CDs implement restrictions that make the game functional only in certain geographic areas) and create backup copies on CD-R and DVD-R media. Although it is legal in some countries for the purchaser to make a back-up copy in case the original is lost or damaged, game consoles contain protection mechanisms that prevent the user from playing the copies. The mod chip makes it possible to bypass these protection mechanisms by supplying the appropriate information. Currently, this is a very common attack mechanism for popular console systems such as the Xbox** and PlayStation** 2. In fact, Microsoft has taken action to prevent those who have modified their consoles from Xbox Live** play[7] (online environment). When a user logs onto an online gaming forum, his or her system is checked for the presence of mod chips. If mod chips are detected, the unit's serial number is recorded, and the device is permanently banned from the network. As a counterattack, mod chips are being produced that can be temporarily disabled to prevent detection.

Software-based protection techniques

The success of online gaming has led to a new set of concerns for the industry. These concerns revolve around maintaining a fair and consistent gaming environment in which players will continue to participate. If some players are able to modify their games, for example, by making their character immortal, the entire gaming experience can suffer. One technique that has been explored by researchers and that could be used to aid in the prevention of game modifications is *code obfuscation*,[8–17] a technique to protect a secret in the application code. The secret may vary; examples are the design of a software component, special algorithms embedded within the software, and important data such as a cryptographic key. Obfuscation works by applying transformations to the code in order to make it more difficult to understand and reverse engineer while preserving the original functionality. The idea is to obscure the readability and understandability of the program to such a degree that it is more costly for the attacker to reverse engineer the program than to simply recreate it or purchase a legal copy. There are three types of obfuscation:

1. Layout obfuscations alter the information that is unnecessary to the execution of the application, such as identifier names and source code formatting (e.g., comments and indentations used for readability).
2. Data obfuscations alter the data structures used by the program. For example, a two-dimensional array could be folded into a one-dimensional array.
3. Control flow obfuscations disguise the true control flow of the application, for example, by inserting dead or irrelevant code or merging multiple functions into one.

The level of protection provided through obfuscation varies with program size and structure. Additionally, obfuscation increases the overhead of the program which could have adverse effects on performance. Since performance is critical in most video games, the degree to which a game can be obfuscated may be limited. A second limitation to the technique is that it does not provide complete protection. Given enough time, a determined adversary will be able to break the protection. However, obfuscation can be used to extend the period of time before the game protection is broken. Because the majority of video games have a short shelf life and most of the revenue is derived over a short period of time, obfuscation is a viable protection technique.

A common feature of many video games is the inclusion of a *license check*. It can be used to verify the validity of the game or to prevent the use of a game after a specific date. To prevent a dishonest player from removing the license check, *software tamperproofing* techniques can be used that prevent the game from being altered.[13,18–23] The tamperproofing mechanism must first detect that the software has been altered. Then, when tampering is detected, the mechanism must cause the program to fail. For the tamperproofing to succeed, the software failure must be stealthy and must not alert the attacker to the location of the failure-inducing code. This can be accomplished by separating the detection and response mechanisms in both space and time. The protection of license checks is just one specific use of software tamperproofing. As with code obfuscation, it can be used to prevent modification of the game executable. The main difference between code obfuscation and software tamperproofing is that code obfuscation is used to hide a secret, whereas tamperproofing is used to prevent alteration of the secret.

One of the most prevalent forms of copy protection for PC-based games is the use of an *installation key*. This is the sequence of letters and numbers generally found on a sticker included with the installation material. During the installation process the user has to enter the installation key in order to verify that the software is a legal copy. This form of copy protection, while common, is generally broken shortly after release. To defeat the system an attacker analyzes the section of code that determines if the sequence of letters and numbers constitutes a valid installation key. Based on the analysis, it is possible to determine the properties a valid key must possess. Once the mechanism is defeated, key generators are posted on the Internet, and anyone can obtain a key to be used with illegal copies of the software.

Currently many video game console-based systems make use of their own proprietary CD or DVD format. When the new console-based systems are released, the software is written on CD or DVD formats that standard burners cannot copy. For example the Xbox uses DVD-9 format, which is a single-sided, dual-layer media format. Nintendo Inc. also took this approach with the GameCube** by using smaller than normal discs. This type of protection technique is usually effective against the occasional copier, but it is not normally unbreakable for long. In fact, software exists that makes it possible to copy any game distributed in a DVD format to regular CD-R or DVD-R/+R media. To play these copies on a game console a mod chip is required. However, techniques exist to play games distributed for console systems on a traditional PC with no special hardware requirements.

The following account illustrates the limited protection provided by proprietary formats. The Sony PlayStation Portable (PSP**) handheld gaming device uses a proprietary 1.8 GB Universal Media Disc (UMD) format for distribution of both games and video content. The initial release of the PSP occurred in Japan on December 12, 2004. This was followed by a release in the United States on March 24, 2005. Only six weeks after the United States release and weeks before release in the United

Kingdom, hackers developed a technique to capture the data from the UMD. Because the UMD uses the standard ISO 9669 format, the captured data can be written to a regular CD or DVD.

The most recently deployed anti-piracy mechanism within the gaming industry is Steam, developed by Valve Corporation. Steam is a content-delivery, digital-rights management (DRM), and multiplayer system.[24] Currently, only a few games use Steam technology. To activate a game using Steam technology requires a Steam account and an Internet connection. A boxed copy of the game purchased at a store includes a Steam client and game content, but not the executable required to play the game. When the game is activated through the Steam system, the files necessary to create an executable game are downloaded. After the initial activation the user can play in single-player mode or on a LAN without an Internet connection. Unfortunately, configuring the game for such activity is not straightforward for the user. Moreover, to participate in online multiplayer games or receive game patches, the user must log in to the Steam system. This scheme makes it possible for Value Software to periodically authenticate the majority of the distributed game copies and to restrict the functionality of illegal copies.

USING SOFTWARE WATERMARKING TO COMBAT GAME PIRACY

One of the difficulties associated with the issue of video game piracy is enforcing the law. Not only can it be difficult for small game developers to prove authorship if their intellectual property is stolen, but it is usually extremely difficult to trace the source of an illegal distribution. Through the use of software watermarking it is possible to address these difficulties.

Fundamentals of software watermarking

Software watermarking takes the approach of discouraging piracy through the attachment of an identifying mark (the "watermark").[25–38] The most basic software watermarking system consists of two functions: $embed(P, w, k) \rightarrow P'$ and $recognize(P', k) \rightarrow w$. Using the secret key k, the *embed* function incorporates the watermark w into a program P, yielding a new program P'. The *recognize* function uses the same key k to extract the watermark from a suspected pirated copy.

A watermarking algorithm is categorized based on a set of characteristics,[39] such as whether the code is analyzed as a static or dynamic object, the type of recognizer used, the embedding technique, and the type of mark embedded.

- *Static/dynamic*—Strictly static watermarking algorithms only use features available at compilation time for embedding and recognition. On the other hand, strictly dynamic watermarking algorithms use information gathered during the execution of the program. Strictly speaking, abstract watermarking algorithms are neither static nor dynamic. Instead, such techniques are static in that recognition does not require execution of the program, but they are dynamic in that the watermark is hidden in the semantics of the program.
- *Recognizer type*—A watermark recognizer is categorized based on the information needed to identify the watermark. Both blind and informed watermarking algorithms require the watermarked program and the secret key to extract the watermark. An informed technique additionally requires an unwatermarked version of the program, the embedded mark, or both.
- *Embedding technique*—To incorporate a watermark, a program has to be manipulated through semantics-preserving transformations. Such transformations can be categorized as follows:
  - — Reorder or rename the code section.
  - — Alter the program's semantics by inserting new nonfunctional code or code that is never executed.
  - — Manipulate the program's statistical properties, such as instruction frequencies.
  - — Alter the program's semantics by incorporating watermark-generating code that directs program execution.
- *Mark type*—An authorship mark (AM) is embedded in every copy of the program and is used to identify the author. It is in essence a copyright notice. A fingerprint mark (FM) is unique for each copy distributed and is normally used to identify the purchaser. Through the use of an FM it is possible to identify the source of an illegal distribution.

Piracy is confirmed by proving the program contains the watermark upon obtaining a suspected illegal copy. From a legal perspective, to prove ownership

it is not sufficient to simply recover the mark from the program. An ideal authorship mark possesses some mathematical property that allows for a strong argument that it was intentionally placed in the program and that its discovery is not accidental. Choosing w such that $w = pq$ where p and q are two large primes is one possible example of a strong watermark. Because factoring is a "hard" problem, only the person who embedded such a watermark would be able to identify the factors p and q.

In order for watermarking to be a viable option for game software, the watermark must withstand attacks against it. There are four types of such attacks: additive, distortive, subtractive, and collusive.

1. *Additive*—In an additive attack an adversary embeds an additional watermark so as to cast doubt on the origin of the intellectual property. Although an attack may succeed even if the original mark remains intact, it is more effective if it damages the original mark.
2. *Distortive*—In a distortive attack, a series of semantics-preserving transformations are applied to the program in an attempt to render the watermark useless. It is the goal of the attacker to distort the software in such a way that the watermark becomes unrecoverable, yet the program's functionality and performance remain intact.
3. *Subtractive*—In a subtractive attack, the attacker attempts to remove the watermark from the disassembled or decompiled code. If the watermark has poor transparency, an attacker may be able to discover the location of the watermark after manual or automated code inspection and then remove it from the program without destroying the software.
4. *Collusive*—In the collusive attack, which is used against fingerprinted software, an adversary obtains multiple, differently fingerprinted instances of a program and is able, by comparing them, to isolate the fingerprint.

Many of the known watermarking techniques are not robust enough to prevent piracy because an attacker can use very simple reverse-engineering tools to identify and remove the mark. The method described in the next section is more resistant to reverse engineering than current methods.

The branch-based watermarking algorithm

The basic idea of the branch-based software watermarking algorithm is centered around the use of a branch function specifically designed to generate the program fingerprint as the program executes. If the branch function is properly designed, the branch-based algorithm can simultaneously embed authorship and fingerprint marks. With many other algorithms, inserting a second mark will destroy the first one. Additionally, tamper detection can be incorporated. We describe the algorithm next and illustrate how these three features can be incorporated in a single branch function.

Any software watermarking system consists of two functions: *embed* and *recognize*. The *embed* function for the branch-based algorithm has four inputs and two outputs.

$$embed(P, AM, key_{AM}, key_{FM}) \rightarrow P', FM$$

Using the two secret keys key_{AM} and key_{FM}, the *embed* function incorporates the authorship mark *AM* into a program *P*, yielding a new program P' and the fingerprint mark *FM*. Because the algorithm can simultaneously embed an authorship and a fingerprint mark, two secret keys are required. The key_{AM} key is tied to the authorship mark and is the same for every copy of the program. The key_{FM} key is required for the fingerprint mark and should be unique for each copy. A fingerprint mark for a particular instance of a program is based on the fingerprint key and the program execution. Thus, the actual fingerprint mark is generated during embedding and is an output of the embed function.

Similarly, the *recognize* function has three inputs and two outputs.

$$recognize(P', key_{AM}, key_{FM}) \rightarrow AM, FM$$

Because the recognition technique is blind, the authorship and fingerprint marks can be obtained from the watermarked program by providing the two secret keys.

The branch-based watermark is dynamic, thus one of the secret keys, key_{AM}, is actually an input sequence to the program. For example, suppose we wish to watermark a tic-tack-toe program; the secret input sequence could be the sequence of mouse clicks that select "X-O-X" on the diagonal. By executing the program with the input sequence, a

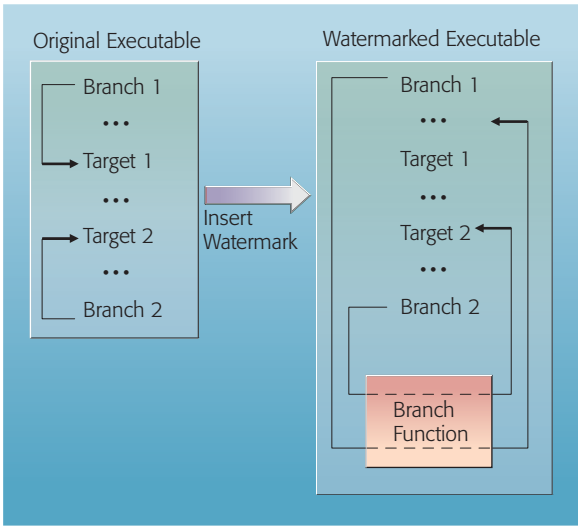

Figure 1
Change in execution flow associated with a branch function

trace consisting of a set of functions F is identified. The set F consists of those functions that participate in the fingerprint calculation. The secret input serves two very important functions in the algorithm. First, it provides a stronger argument for the validity of both the authorship and the fingerprint marks. Second, the secret input makes recognition more reliable. Only when the program is executed using the secret input can it be guaranteed that the fingerprint is generated.

Fingerprint branch function

The fingerprint mark for a program is generated as the program executes through a specifically designed branch function. We call this function a *fingerprint branch function* (FBF). The FBF is an extension of the branch function originally proposed by Linn and Debray as part of an obfuscation technique used to disrupt static disassembly of native (x86 compatible) executables.[17] The obfuscation converted unconditional branch instructions to a call to a branch function inserted in the program. The sole purpose of the branch function was to transfer the control of execution to the target of the unconditional branch. ***Figure 1*** illustrates the change in the execution when a branch function is used to control the execution behavior. The branch function can be designed to handle any number of unconditional branches. To increase the versatility of the branch function we have devised an extension that makes it possible to convert conditional branches as well. When this idea is applied to the x86 microprocessor architecture instruction set, all *jmp*, *call*, and *jcc* instructions can be converted to calls to a single branch function.

In order to provide fingerprinting and tamper detection capabilities, the branch function is enhanced. The original branch function was designed simply to transfer execution control to the branch target. In addition to the transfer of control, the FBF also incorporates an integrity check and key evolution. The FBF performs the following tasks:

- An integrity check producing the value v_i.
- Generation of the next function key, k_{i+1}, using v_i and the current key k_i, $k_{i+1} = g(k_i, v_i)$.
- k_{i+1} is used to identify the return instruction.

Each time the FBF is called, a new key, k_i, is calculated. The k_i key is then used to aid in identifying the original branch target. There are a variety of ways the FBF can be implemented. Many of the details are specific to the particular architecture. For example, when the FBF is implemented for use in native executables, one possible way to link key generation to program execution is to use the key to identify the displacement to the branch target. In this case the branch-target displacements associated with each replaced branch are stored in a table. The key is used to access the table. The displacement obtained is added to the return address that was stored on the stack when the FBF was called. After the execution of the FBF is completed, the next instruction to execute is the target of the original branch instruction. This is contrary to the traditional assumption that a function returns to the instruction following the call instruction.

The original branch function obfuscation can provide minimal tamper detection. A table is used to store a displacement. Therefore, any transformation applied to a function which alters the displacement between a branch and its target causes the branch function to return to an incorrect instruction. A key aspect of the FBF is the use of integrity checks, which make it possible to extend the tamper detection capabilities throughout the entire program. An integrity check is an inserted section of code used to verify the integrity of the program and to identify active debugging.[40] Integrity checks are

triggered during software execution. For example, one of the integrity checks could choose a block of code and calculate its checksum. If the attacker attempts to store break points or modify the code, even if the modification is very slight, the checksum will be incorrect. When tying to detect the presence of a debugger, the elapsed time of execution from one point to another can be used as an integrity check. These simple integrity checks are just for illustration purposes. In practice, a variety of stealthy integrity checks are used. Often these checks are customized to address the specific requirements of the application. One of the limitations of the use of integrity checks is that describing them in detail decreases their potency. This is true of most techniques aimed at providing tamper resistance.

In addition to fingerprint generation and tamper detection, an authorship mark can be incorporated in the FBF. One possible means of encoding the mark is to choose a one-way function such that one of the variables can be set to *AM*. For example:

$$k_{i+1} = \mathrm{SHA1}[(k_i \ \mathrm{XOR} \ AM) \,||\, v_i]$$

A one-way function is a function in which it is easy to compute $y = \mathrm{f}(x)$, but given y, it is difficult to reverse the process and find x. SHA-1 is one example of a one-way function (SHA-1 stands for Secure Hash Algorithm).

Embedding the fingerprint branch function in the executable

The branch-based watermarking algorithm is a general scheme that can be implemented in a variety of different ways. Many of the implementation details are architecture specific; however, variations can be constructed based on the architectural specifications. In general, the embedding of the authorship and fingerprint marks occurs by embedding the FBF into the program and converting branch instructions to calls to the FBF. The following algorithm illustrates a possible embedding transformation for watermarking native executables. In this technique watermarking is accomplished by disassembling a statically linked binary, modifying the instructions, and then rewriting the instructions to a new executable file.

The embedding process consists of three phases. In the first phase, an execution trace of the program is obtained based on the secret input sequence. The trace identifies a set of functions F through which execution passes. These functions will be modified to incorporate the fingerprint generating code.

In the second phase of the algorithm, the branches in each function f in F are replaced by calls to the FBF. Special care must be taken in selecting which branch instructions are converted. The branch instruction used for the fingerprint computation must reside on a path through the function that will be traversed every time the function is executed. Without this constraint, an irregular key evolution would occur, which would result in transferring execution to an incorrect instruction. This would ultimately lead to improper program behavior. In addition, because a new key is generated every time the branch function is executed, the branch instructions cannot be part of a nondeterministic loop. Thus, all branches along the deterministic path through the function are replaced with calls to the FBF. In order to identify the deterministic path, we compute the dominator set for the exit block in the function control flow graph. The dominator set may include blocks that are part of a nondeterministic path such as a loop header. Any such block is removed from the path.

For each branch replaced, a mapping between the calculated key and the branch-target displacement is maintained.

$$\theta = \{k_1 \rightarrow d_1, k_2 \rightarrow d_2, \ldots, k_n \rightarrow d_n\}$$

Because the key is paired with the displacement at the time the branch instruction is replaced, the instructions must be replaced in execution order. This is again addressed by using the dominator set for the exit block.

θ is used in phase three to construct a table T, which is stored in the data section of the binary. The table is used to store the branch-target displacement for each branch in the program that has been replaced. The first step in laying out the table is to construct a perfect hash function[41,42] such that each key maps to a unique slot in the table. It is best to use a minimal perfect hash function so that the table size is minimized.

$$h : \{k_1, k_2, \ldots, k_n\} \rightarrow \{1, 2, \ldots, m\}, n \leq m.$$

The displacements are stored in the table such that $T[h(k_i)] = d_i$.

The fingerprint branch function is a new function inserted in the program during embedding. The function is constructed such that the following tasks are performed:

- An integrity check producing the value v_i.
- Generation of the next function key, k_{i+1}, using v_i and the current key k_i, $k_{i+1} = g(k_i, v_i)$.
- Identification of the displacement to the next instruction via $d_i = T[h(k_i)]$, where T is the table stored in the data section and h is a hash function.
- Computation of the return location by adding the displacement d_i to the return address.

Unlike the authorship mark, the fingerprint mark is not embedded in the program. Instead, it is generated as the program is executed. Each function in the set F obtained by executing the program with the secret input sequence produces a final function key. The keys are combined in a commutative way (e.g., add the values) to produce the fingerprint mark for the program.

The variation in the fingerprint mark is obtained through the fingerprint key, key_{FM}, which is unique for each copy of the program. The key_{FM} key is used to begin the key evolution process in each fingerprinted function. Based on the unique key, the fingerprint for each program will evolve differently. Because the key is used to access the inserted structure, each program will contain a differently organized structure. It is important that key_{FM} is available each time the program executes. There are a variety of ways this can be accomplished. For example, it could be embedded in the program, or it could be required that the user enter it each time the program is started. To prevent a user from using the initial key in an attack, secure computing devices such as the TPM available in the ThinkPad** laptop could be leveraged.

Recognition

As with embedding, the first step in recognizing the embedded marks is to execute the program using the secret input. The execution will identify the set of functions F, which have been fingerprinted, as well as the FBF itself. Once the FBF has been identified, the one-way function to extract the authorship mark can be isolated. To extract the fingerprint mark, the location where the final function key is stored for each f in F must be accessed while the program is executing. The final function keys are combined to form the fingerprint mark.

Registration-based customization

The only static variation in differently watermarked instances of a program is in the inserted structure. This feature enables software companies to produce and distribute fingerprinted software in the traditional manner. The program purchased would be nonfunctional until the user installs the software and registers it with the company. Upon registration, the user key and structure is distributed, creating a fully functioning program. Previously, if a software company wanted to tie a specific fingerprint mark to a purchaser, the user had to purchase the software directly from the company, and the program was fingerprinted at that point. By using the branch-based watermark, distribution of fingerprinted software can be accomplished through prepackaged software sold at retail stores. Installation of a fully functioning copy does require an initial Internet connection; however, Valve's Steam technology has demonstrated that a required connection may no longer be a drawback.

One important distinction to make between the branch-based software watermarking technique and fingerprinting techniques used for media is that the technique is not based on signal processing. A media fingerprint is often embedded by the media player. This makes the technique vulnerable to an attack in which the media player is prevented from actually embedding the mark. In the event of such an attack, the non-fingerprinted media is still playable. When a piece of software is prepared for fingerprinting using the branch-based technique, the proper control flow is removed. The control flow is added back into the program when the fingerprint is embedded because the execution behavior is tied to the generation of the fingerprint. If an attacker blocks the embedding of the fingerprint, the program is nonfunctional.

Evaluation

The branch-based watermarking technique was proposed by Myles and Jin.[3] They performed a thorough evaluation with respect to robustness against attack and overhead incurred. In this section we summarize those results so as to demonstrate the viability of the branch-based watermarking algorithm for content protection of video games.

The evaluation was performed using the SPEC CINT2000 benchmark suite.[43] To evaluate the

robustness of the algorithm, four categories of attacks were examined: additive, distortive, collusive, and subtractive. For an additive attack to be successful the program has to continue to function properly after the embedding of the second watermark. To simulate an additive attack, the test applications were double watermarked using the branch-based algorithm. This resulted in improperly functioning applications. Similar results could be obtained by using other watermarking algorithms in the attack. The attack fails because the integrity check detects the program alteration.

In a distortive attack, the goal of the attacker is to distort the software such that the watermark is unrecoverable, yet the program's functionality and performance remain intact. To verify that the branch-based algorithm is resistant to distortive attacks, five different obfuscations were applied to the watermarked applications. In each case the alterations were detected, resulting in improper functionality.

The collusive attack is the most crucial for fingerprinted software. Previous watermarking algorithms have relied on the use of obfuscation to prevent a collusive attack. The general idea is to apply different sets of obfuscations to the fingerprinted programs so that they differ everywhere. The branch-based algorithm is resistant to the collusive attack even without the use of obfuscation. Two differently fingerprinted programs differ only in the order of the values in the table added to the data section of the binary. Examining the code segment of the application does not aid the attacker.

Because an attacker has full control over the software, without the use of a completely secure computing device, guaranteed protection against a subtractive attack is not possible. Instead, the goal is to design a technique in which the analysis required to remove the watermark is too costly. The robustness of the branch-based algorithm is partially dependent on the number of branches that contribute to the fingerprint calculation. By requiring the branches to be on a deterministic path, the number of usable branches is decreased. Through an analysis of a variety of different applications, it was found that a satisfactory number of branch instructions exist. To remove the watermark, the attacker has to identify the sections of code generating the fingerprint and patch the executable. Such an attack first requires identifying each of the converted branch instructions. The second step involves identifying the correct target for each branch. Finally, each of the call instructions must be replaced with the correct branch (which could be a *jmp*, *jcc*, or *call* instruction) and displacement. For the attack to be successful, all converted branches must be identified and replaced. Although such an attack is not impossible, the manual analysis required to accomplish such a task is extensive, especially because many analysis tools can be thwarted through the integrity checks. Additionally, the attack can be further complicated by the incorporation of the strength-enhancing features described in the original paper.[3]

Table 1 Effect of watermarking on program execution time and size

Program	Branches Used	Slowdown	Size Increase
gzip	79	1.00	1.04
vpr	405	1.00	1.19
mcf	24	1.00	1.06
crafty	94	1.00	1.01
parser	239	1.13	1.02
gap	742	1.00	1.18
vortex	477	1.00	1.09
bzip2	135	0.99	1.09
twolf	233	1.01	1.05

The cost incurred due to watermarking was also evaluated using the SPEC CINT2000 benchmark suite. As can be seen in ***Table 1***, very little performance overhead was incurred by the additional calls and integrity checks. Only one application (*parser*) suffered a noticeable slowdown of 13 percent. The performance of the other benchmarks was between 99 and 101 percent of the original. The impact on size was a bit more noticeable with increases between 1 and 19 percent. Since the fingerprint is generated as the program executes, the size of the fingerprint does not impact the size of the watermarked program. The majority of the size increase is a result of the table inserted in the data section and the size of the inserted fingerprint branch function. For each branch replaced, a single slot in the table is required. Using a straightforward implementation of the algorithm, each slot is 4 bytes. Thus, the minimum table size is (num branches $\times$ 4 bytes). Through the use of a minimal perfect hash function the number of empty slots can be minimized. For most applications, the size

increase was minimal. Additionally, the implementation used to generate the results did not use a minimal perfect hash function; thus, the results could be improved.

CONTENT PROTECTION FOR PHYSICAL MEDIA: AN INTRODUCTION

The factors that led to rampant video game piracy have also created similar issues for the distribution of premium entertainment content. Due to these similarities, it is natural to apply the concepts developed for the protection of entertainment content to video games. More specifically, parallels can be drawn between the protection of console games and that of audio or video content on prerecorded media, such as the video content on DVDs. Similarly, the concepts for the protection of software media players can be applied to PC games. To the best of our knowledge, such parallels have yet to be leveraged in developing protection technologies for games.

In this section we briefly review the evolution of copy protection technologies developed for prepackaged audio/video content and for recordable media. In particular, we focus on broadcast encryption, a cryptographic-key management technology that has turned out to be well suited for this application domain.

Copy protection

One of the first and most widely known encryption-based protection schemes for optical media is the content scrambling system (CSS). CSS was introduced in 1997 for DVD-video recordings. The essence of the scheme is the use of a small set of secret, static global keys. The set of keys are shared between the studios and the DVD player manufacturers upon signing of the CSS license. DVD-video content is encrypted with the global keys during the DVD mastering process. The content can then be decrypted by all licensed players that employ one of the global keys.

CSS is not a true copy protection system—no attempt is made to prevent identical copies from working. A bit-for-bit copy of a CSS-protected DVD still plays in all licensed players. The purpose of CSS is to limit playback of DVD-video recordings to licensed players.

The first successful attack against CSS was launched in 1999. It was based on obtaining one of the secret keys by reverse engineering a licensed software player. Because the cryptography used in the system was weak, the remaining secret keys were discovered using cryptanalysis. These attacks enabled the development and distribution of non-licensed players such as the DeCSS software. An obvious consequence of using a global secret-based key management scheme such as CSS is that once the system is compromised, there is no recovery without updating the entire player population.

Broadcast encryption

When CSS was introduced, the global secret-based key-management system was thought to be a necessary consequence of having to distribute encrypted content to a large number of disconnected player devices. However, unbeknownst to the developers of CSS, Fiat and Naor had developed a key-management scheme in 1993 specifically for one-way communication channels, called *broadcast encryption*.[44] Originally designed to address the problem of renewability for conditional access applications, broadcast encryption was, several years later, successfully applied to the problem of protecting content on physical media. This new direction of copy protection leverages the fact that DVD players are not completely disconnected devices—they have a one-way connection to the publishers, who are continuously providing a stream of new content.

Fundamentally, a broadcast encryption system is designed to encode a management key that is to be transferred from a sender to a multitude of receivers. This management key is typically used as an encryption key for the protection of some payload data. The purpose of the broadcast encryption scheme is to be able to efficiently modify the encoding of the management key such that one or more receivers are excluded from receiving management key updates. This prevents the affected receivers from continuing to decrypt the payload data. The key management information, which encodes the management key, is referred to as the key management block (KMB). The broadcast encryption keys stored by the receivers are called device keys (DK).

To illustrate, consider the very simple broadcast encryption scheme in ***Figure 2***. In this scheme, each receiver has a single unique device key k_i in $\{k_1 \ldots k_n\}$. The management key K is encrypted once

for each potential receiver, $e(K)k_i$ (this notation signifies that we are encrypting management key K using device key k_i). To obtain K, a particular receiver needs to locate the corresponding entry in the KMB and decrypt the management key using its device key. To revoke a particular receiver, the KMB simply needs to be updated to remove the corresponding entry.

Some of the advantages of broadcast encryption are obvious, even with this simplified model. It is very efficient. All encryptions are performed by using symmetric algorithms such as the Advanced Encryption Standard and are therefore very fast. To obtain the management key, a receiver has to process some index information and perform a single symmetric decryption.

The simplified scheme suffers from a significant flaw: the KMB is prohibitively large. Suppose storing a single encrypted key takes 20 bytes and the system has to support one billion receivers; then, the size of the KMB would be 20 GB of data. This makes the scheme completely impractical for almost all applications. Fortunately, techniques exist for significantly improving the efficiency.

In *Figure 3* the simple broadcast encryption scheme is extended by giving each receiver an additional key. This key is shared with half of the device population. In this example, assume that all receivers with an even serial number share a copy of the key k_e, and all receivers with an odd serial number share a copy of the key k_o. Thus, each receiver has two device keys—an individual device key and a copy of either k_e or k_o.

How this modification affects the size of the KMB is illustrated in Figure 3. Initially, all receivers are included, and the KMB is very small. Receivers with an even serial number are using k_e to decrypt the management key K, and receivers with an odd serial number are decrypting the management key with k_o. Thus, if the size of an entry in the KMB is 20 bytes, its size is 40 bytes. Note that this size is independent of the number of receivers; a system that includes a billion receivers has a KMB of 40 bytes.

The improvement quickly degrades, however, as devices are revoked. The second KMB shown in Figure 3 excludes receiver 2 (R2). Because R2 has a copy of k_e, it cannot be used in the KMB anymore. Therefore, all other receivers with an even serial number have to be included in the KMB by encrypting the management key K with their individual keys. Thus, an additional $(n/2) - 1$ entries, where n is the number of receivers in the system, have to be added to the KMB. In the system in Figure 3, which contains six receivers, two entries must be added to the KMB when R2 is revoked. However, if we consider a population of a billion receivers and 20 bytes per entry, the resulting KMB will have a size of 10 GB.

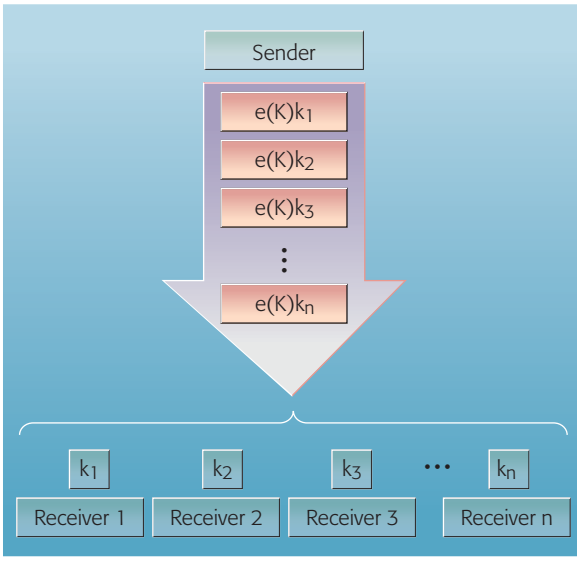

Figure 2
Simplistic broadcast encryption scheme

Continuing with the example, it is interesting to observe the effects of excluding an additional receiver. If the additional receiver has an even serial number, the size of the KMB will not change significantly. However, if the receiver has an odd serial number, the use of k_o has to be discontinued. In this case, the system degrades to the simple model in the initial scheme, and the size of the KMB, assuming a system with a billion receivers, increases to approximately 20 GB. This example highlights an important trade-off for this family of key-management systems: the size of the KMB can be reduced at the cost of increasing the number of device keys. Also, we can see that it is desirable to start out with a very small KMB, which grows in a linear relationship to the number of excluded devices.

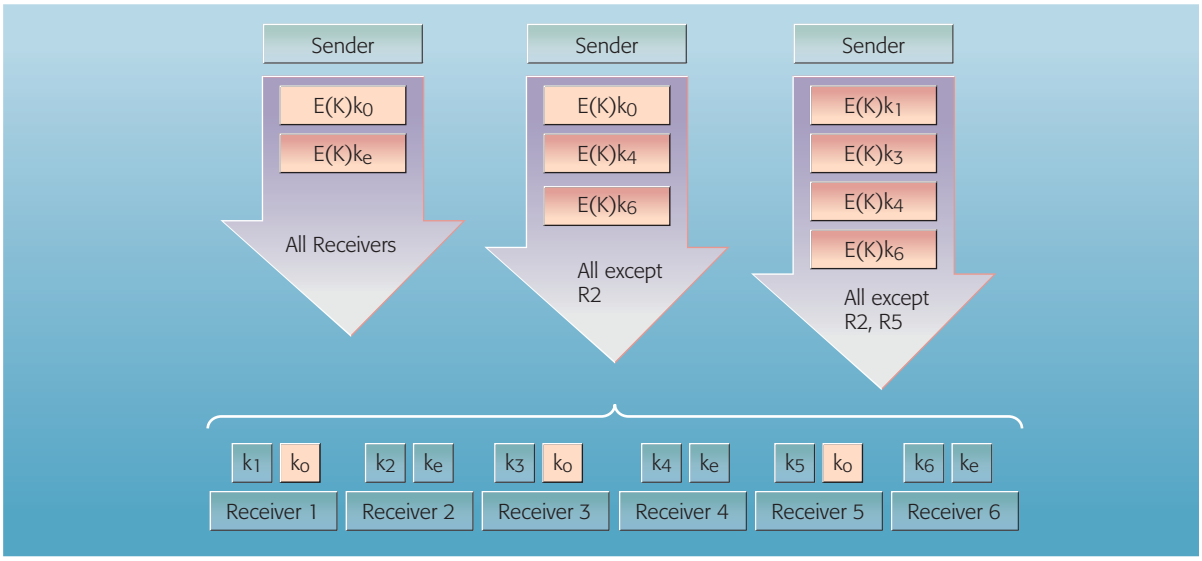

Figure 3
Simplistic broadcast encryption scheme with two device keys

Currently, broadcast encryption is used in at least four copy protection systems for media. The first such scheme, CPRM, is based on a system of keys arranged in the structure of a matrix. A device has 16 device keys. Although each individual key might be shared with some of the other devices, the combination of 16 keys is unique. CPRM is used on various formats with slightly different parameters, which among other things determine the size of the KMB. For example, the KMB used on recordable DVDs is 320 KB. It is of constant size, independent of the number of excluded receivers. It is important to point out that the key management system used by CPRM has finite revocation capabilities—once the maximum number of revocations is reached, it is no longer possible to exclude an additional receiver without affecting some of the remaining devices in the system. Note that the same restriction also applied to the original scheme presented by Fiat and Naor in 1993.[44]

Shortly after the introduction of CPRM, two very similar broadcast encryption schemes with unlimited revocation capabilities were developed.[45,46] The logical key hierarchy (LKH) scheme uses a tree-based structure of keys. Even though LKH provides unlimited revocation capabilities, it still has approximately the same space efficiency as the matrix-based key management scheme used in CPRM.

Two years later, Naor, Naor, and Lotspiech[47] developed the NNL broadcast encryption system. NLL provides unlimited revocation capabilities, at the same time significantly reducing the size of the KMB. In NNL the keys are arranged in multiple tree structures. Under this scheme, a receiver has approximately 400 device keys. The NNL KMB is about 25 times more concise than that used by LKH. Requiring about 1.28 entries in the KMB per revocation, NNL is nearly as efficient as a public-key certificate revocation list without requiring connectivity or expensive public-key calculations. An NNL KMB starts out very small and grows in a linear fashion as the number of revoked receivers goes up.

Broadcast encryption and protection of content on physical media

CPRM is available on a variety of formats, such as DVD-RAM, DVD-R/W and flash-memory cards like the Secure Digital (SD**) memory card or Secure CompactFlash**. DVD-Audio uses a version of CPRM for prepackaged content which is called CPPM (Content Protection for Prerecorded Media). The 4C specifications for CPRM and CPPM are publicly available and can be downloaded from 4C Entity's Web site.[48] With over 200 million devices enabled and KMBs on every SD memory card, 4C technology is in widespread use today.[49] We now describe how CPRM leverages broadcast encryption for copy protection.

Access authorization

To use broadcast encryption for protecting content on physical media, each player must have its own set of device keys. In the case of prerecorded content, such as DVD-Audio, the media itself contains the KMB, followed by the encrypted content. For recordable media, the KMB is pre-embossed in the disk's lead-in area. Thus, all blank, recordable 4C media available in retail stores contain a current KMB. Additionally, to encrypt the content for storage on recordable media requires the recorders to have a set of device keys. The device keys are used by the recorder to process the KMB to obtain the management key used in encryption.

The system, as described so far, can be used to manage device compliance. Upon signing the 4C license, a device manufacturer can start ordering device keys. If device keys get compromised, as was the case in DeCSS, new media will contain an updated KMB which revokes this particular set of keys. All of the CPRM-enabled recordable media formats are designed to be usable with unprotected content. In this case, recorder and player applications do not require device keys, and the security features of the protected media remain unused. Even if the copy protection features are used, CPRM does not require the application to be authenticated. Any application can retrieve the Media ID and the KMB and read or write the encrypted content—but to make any sense of the content or to produce valid data, the application needs to have device keys.

Copy protection

In order to enable a compliant, unrevoked device to distinguish between the original and a bit-by-bit copy requires the introduction of a unique Media ID. The Media ID is the unique, read-only disk serial number. Combining the Media ID with the management key obtained from the KMB binds the content to a particular instance of physical media. This binding prevents a bit-by-bit copy from playing. Theoretically, a compliant recorder containing a valid set of device keys can create a valid copy. To create the copy, the content is read from the source disk, rebound using the new serial number, and written to the target disk. However, because such a recorder needs to have device keys in order to process the KMBs on the source and target media, it has to be a license-compliant device. Such a device will not create a copy unless allowed by the usage conditions associated with the content. The revocation capabilities of the underlying broadcast encryption scheme make this rule enforceable, which prevents licensed manufacturers from building an illegal copying device.

To cleanly separate the binding step from content encryption and to simplify legitimate rebinding in case of a copy or move, another level of indirection is introduced: the management key derived from the KMB combined with the disk serial number yields a media unique key. The media unique key is not used to encrypt the bulk content on the media. Instead it encrypts the title keys which in turn can be used to access a title (a title is a self-contained unit of content such as a complete motion picture). This allows content to be copied or moved to a different medium simply by re-encrypting the title keys with the new media unique key.

Management of usage conditions

In order to protect sensitive data from being modified by unauthorized applications, an additional protection mechanism is needed. This requirement exists for both static data, such as usage conditions, and dynamic data, such as the actual value of usage counts or time stamps. All CPRM media manage usage conditions by combining their hash with the title key calculation. The simple extension of the binding calculation ensures that if the usage restrictions are modified by an unauthorized application, any subsequent decryption of the protected content will fail.

This approach does not work, however, for dynamic usage data on a recordable medium that needs to be updated. In that case, an attacker can do a complete backup of the medium and restore it, for example, after a usage count is decremented. To address this requirement, CPRM-enabled drives support a mechanism for secure state management. This is either done with a two-way authentication protocol between drive and application or by having the drive generate a cryptographic nonce (an arbitrary number used only once in a security session) that is used to protect the sensitive data against unauthorized backup and restore. The effect is the same—only a licensed application that has device keys is able to make updates to this authenticated area.

Drive authentication

For software players, the architecture as described in this section remains essentially the same. A software application running on a general-purpose

computing device needs to have device keys in order to access protected content on CPRM-enabled media. Nevertheless, there is an attack specific to this architecture. Because the application is running on an extensible platform, there is a risk when the drive containing the protected media is virtualized. On an open platform with an extensible driver model, the application needs to have a way to tell that it is really interacting with a disk drive containing the protected media, instead of a device driver that merely replays the stream of encrypted information that a valid drive would originate. To solve this problem, CPRM-enabled drives support a drive authentication mechanism that uses a challenge-response protocol which allows the application to authenticate the drive. By introducing a randomly generated challenge, the physical drive can no longer be "spoofed" (have another drive masquerade as this drive). Different CPRM-based formats use different drive authentication protocols—whereas some take advantage of the broadcast encryption key material, others are based on separate keys.

Summary

The broadcast encryption-based copy protection schemes for physical media provide several features for the protection of premium content:

- *Authorized access*—Protected content is accessible only to authorized devices and applications that have valid device keys.
- *Copy protection*—The protected content is strongly associated with a particular instance of media and its unique Media ID.
- *Management of usage conditions and related state information*—This allows licensed devices and applications to securely manage copy control information and related state data, such as the maximum number of plays and the number of plays already consumed.
- *Drive authentication*—On open platforms such as PCs, there is an additional protocol allowing the compliant application to authenticate the drive. This prevents drive spoofing and ensures possession of the media.

These features of copy-protected media are as important for the protection of games as they are for the protection of audio/video content. As we will show, both PC-based and console games can benefit from them. Before we focus on applying these protection capabilities to games, we will introduce another interesting aspect of the broadcast encryption-based copy-protection scheme described in this section: it can be used in a server-side binding model to enable electronic distribution of content to physical media.

Electronic distribution of protected content

The obvious way to add secure electronic distribution capabilities to CPRM-enabled media is with a DRM. However, if there is no need to manage content on the PC, there is an easier way. The copy protection architecture described in the last section can be extended to provide protection for content during online distribution. In 2004, 4C Entity published a specification for this architecture, calling it CPRM for Network Download.[50]

In this model, the binding operation during the recording process is performed across a network connection to a server system. As we discussed in the previous section, the binding operation involves processing the KMB of the target media with the recorder's device keys to obtain the management key. Then the management key is combined with the Media ID of the target media to calculate the media unique key. Finally the title keys of the content are encrypted with the new media unique key. These are the steps that a CPRM-compliant recorder performs to create a valid recording.

In the network download model, illustrated in ***Figure 4***, the binding operation is now performed by two components instead of one—a lightweight client application that runs on a PC in the end user's home and a remote license server that has access to device keys.

In this model, the client application interacts with the drive to read from and write to the CPRM-enabled media. The license server performs the cryptographic binding calculation.

The client application retrieves the Media ID and the KMB from the target media (step 2 in Figure 4) and transfers it over the network to the license server (step 3). The license server uses its device keys to perform the binding operation and sends the encrypted title keys back to the client (step 4), who writes it onto the target media (step 5). At this point, the recordable media is ready to receive the

encrypted titles, which can be downloaded by the client application in any arbitrary way and written onto the target media.

There are different ways to communicate the end user's content selection to the license server—the model shown in Figure 4 assumes that there is an eCommerce service involved that guides the end user through the online shopping experience and ensures payment has been obtained. At the end of this process, the eCommerce service will launch the client application and pass it a content identifier, Content ID, in order to begin the server-side binding process (step 1 in Figure 4). Also, the CPRM license server has to be able to obtain the title key corresponding to the content that is purchased. These keys could be managed in a database at the license server. Note that for this reason the license server also needs to receive the Content ID in step 3 so that it can pick the corresponding title key for the binding operation.

The server-side binding model has a number of interesting characteristics. Most important, there is no requirement to manage sensitive key material in the client application. In fact, this client could be open sourced without compromising the security of the system. The license server, on the other hand, can be hosted in a secure facility to provide adequate protection for the device keys. Also, the CPRM download architecture has excellent privacy properties. The security model itself does not force the end user to reveal his or her identity, and it does not even allow the license server to build a profile of the consumer because it cannot tie individual transactions together. Of course, the payment or subscription component in the eCommerce service might require identification—but if an anonymous subscriber management system is used, the transaction can be kept completely confidential. Finally, observe that the content itself remains persistently encrypted and can be obtained by the client application in whatever way desired. Because all media-specific information is contained in the protocol flow with the license server, the encrypted content object can be cached or even shared among users in a peer-to-peer fashion if desired.

CONTENT PROTECTION FOR GAMES: DEPLOYMENT SCENARIOS

In this section we present five scenarios that illustrate how the previously described protection mechanisms can be deployed for video-game copy protection. In each of the scenarios we discuss how the proposed mechanisms address the attacks described in the section "Gaming and piracy." In addition, PC and console-based gaming have differing capabilities, and therefore, the copy-protection mechanisms must be customized to the particular platform. In these scenarios we illustrate how the described protection mechanism can be customized to address platform differences. Lastly, the scenarios illustrate how the protection mechanisms do not restrict the method of distribution; that is, they apply both to physical media and distribution by download.

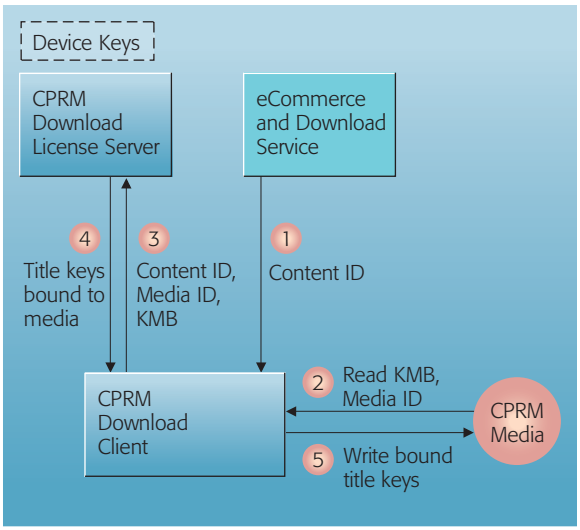

Figure 4
CPRM for network download

Watermarking for PC-based games

Piracy protection for PC-based games has the advantage that protection technologies developed for general software can be directly applied. By using the software-watermarking technique previously described, each individual copy can be customized to the user. Using this model, software can either be distributed as prepackaged software purchased at a retail store or through online download distribution. In either case the game package contains the installation executable as well as a crippled game executable, which is nonfunctional until the installation process has been completed. During installation, a one-time connection to the registration server is required. The registration process requires that a user submit some form of unique identification in exchange for the water-

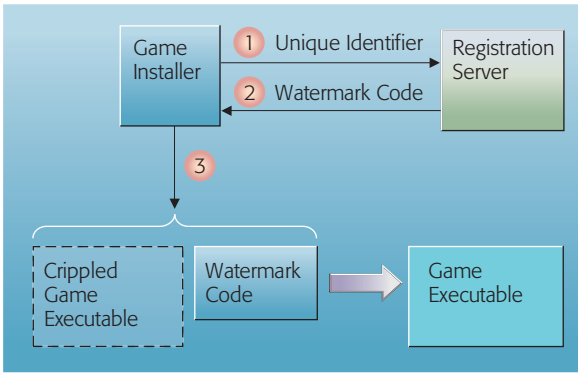

Figure 5
Installation and registration of a watermarked game

mark-specific code, which creates the fully executable game linked to the user. The unique identifier must enable the future identification of the user. Additionally, this information has to be personal enough that the user is unlikely to share it with others. An example of such information includes a credit card number. *Figure 5* illustrates the installation process of a watermarked game that is linked to the user.

Using this protection model, the user is not prohibited from playing the legally registered game on multiple machines. This allows for the portability desired by the gaming community, which is generally accomplished by transporting the physical media. The model also does not prohibit the player from creating back-up copies of the game in countries where such activity is legal. However, this model does address the difficulty associated with identifying the source of an illegal distribution. When an illegal copy of the game is produced and distributed, the copy contains the watermark associated with the original owner. When illegal copies are recovered, the watermark is extracted and compared with the data contained on the registration server. This information leads to the user who is guilty of illegally redistributing the game.

Copy protection and watermarking for console games

In this model, we apply the copy protection technology described in the section "Content protection for physical media" and our software watermarking capabilities presented in the section "Using software watermarking to combat game piracy" to the problem of copy protection for console games.

This is a straightforward application of CPRM and CPPM to gaming consoles. In this model, the console is equipped with device keys that have to be robustly embedded in the device. The media used to distribute the gaming content uses broadcast encryption-based copy protection as described in the section "Content protection for physical media." It comes with a KMB that is capable of revoking compromised game consoles or any applications that copy gaming content off a disk for redistribution.

Figure 6 illustrates a way to combine copy protection functionality with our watermarking technology. This assumes that the game console has connectivity and is capable of persistently managing the fingerprinted executable or at least the fingerprint data in permanent local storage. Whenever a new game is first played on the console, the console has to go through the fingerprint registration flow, and the game is digitally fingerprinted (side marked "Play"). The registration information submitted for fingerprinting can include both a personal identifier that the user is reluctant to divulge publicly and a console identifier. Should the content be pirated, this allows the identification of the compromised console and the subsequent revocation of its device keys with an updated KMB.

As Figure 6 shows, this basic model can be combined with the CPRM download architecture described in the section "Content protection for physical media" to enable electronic distribution of new gaming content by means of a PC-based client application (side marked "Distribution"). Following the CPRM for Network Download model, there is no need to hide sensitive key material on the PC because the cryptographic binding calculation is done by the license server. The result of the CPRM download is a CPRM-enabled disk that contains protected content which is in every aspect similar to a prerecorded disk available in retail stores.

The security properties of this model are similar to those of a player for 4C-enabled content, except that the fingerprint provides an additional level of protection. Attempts to copy the installation media or to lift the content off the installation media are thwarted by the copy-protection technology. If the attack is focused on the console, the attacker might

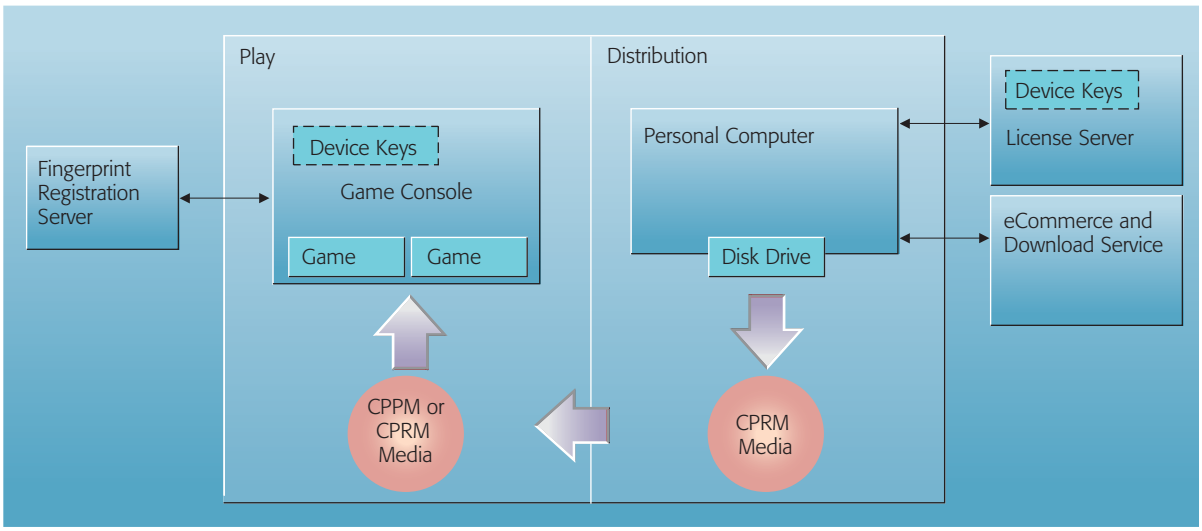

Figure 6
Copy protection for console games

succeed in obtaining a decrypted copy of the game but will still have to overcome the digital fingerprint in order to avoid being tracked down. Should the attacker decide to redistribute the fingerprinted games, we have two options. Using the personal identification obtained during the registration process, legal steps can be taken against the attacker. Also, based on the console identifier, we know what device keys the attacker is using, and we can revoke them in subsequent game releases—either via electronic download or distribution on prerecorded media.

The most difficult to defend against is an attacker who managed to obtain valid device keys and is therefore in a position to create and distribute new installation images on recordable disks. With our current fingerprint technology there is no way for us to trace the attacker in this scenario because the fingerprint has not been inserted yet. Other forensic technologies, such as traitor-tracing schemes, might be needed to be able to identify the source of the pirated games. Applying these concepts to executable content is certainly an area for future research.

Copy protection and watermarking for PC games

In this scenario we leverage the properties of CPPM to develop an architecture for media-based distribution of PC-based games. We reuse concepts developed for software implementation of 4C media players.

What differentiates the PC from the gaming console in the first scenario is that PCs, or any other general-purpose computing platform, do not come equipped with device keys. To solve this problem, we introduce a game loader component that manages device keys protected by tamper-resistant software and capable of executing encrypted gaming content. The game loader needs to obtain 4C device keys. This can be done in one of two ways: The game loader can be shipped with shared device keys with a limited validity period. That approach requires a solid online update capability so that the game loader can be updated with a new version as soon as the shared keys expire. Alternatively, there can be a registration flow as shown in step 1 in ***Figure 7***. This registration does not require payment but must include a Personal ID suitable to discourage an end user from registering multiple times with different identities in order to get multiple sets of device keys.

The game loader itself can be shipped on the copy-protected disk as an unprotected file to be installed automatically when the disk is inserted in the drive. The loader is a generic component designed to be used with a variety of different titles. It only requires updating if it becomes known that a particular implementation is compromised, and updating can happen either over the network or with new media.

After the game loader setup is complete, the actual game can get installed onto the PC. At this point, the

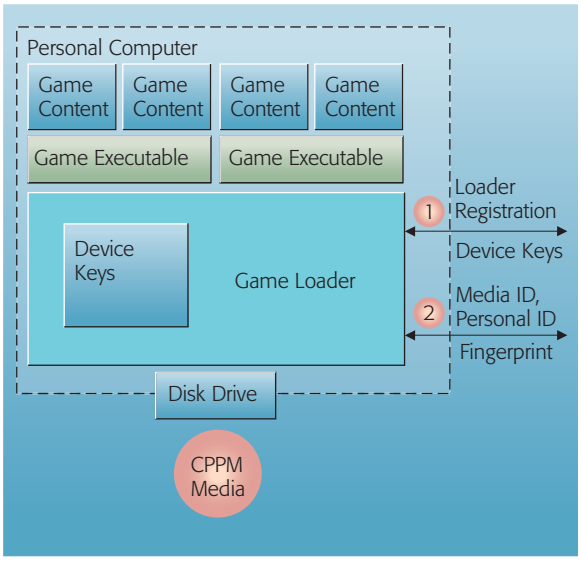

Figure 7
Copy protection and watermarking for PC games

game itself is registered, and user-specific watermark information is inserted into the game executables, thereby creating a functional, fingerprinted, and encrypted binary. As part of the registration flow (step 2 in Figure 7), both the unique Media ID of the installation media and the Personal ID are sent to the registration server in order to prevent a copy from being registered multiple times by different users. The Personal ID has to be an identifier that can be traced back to the user and that the user is reluctant to share with others. This could be a credit card number or a subscriber ID and password combination for an online system operated by the game publisher—as long as that system is based on authenticated users.

The game loader now plays the role of a software player for CPPM-protected multimedia content: To execute a game, the game loader decrypts pages of the fingerprinted executable on the fly and executes the game. On the Windows** family of operating systems, the game loader could be implemented as a file-system filter driver. The game loader can also take advantage of the CPRM drive authentication procedure to make sure the original installation disk is available. Finally, the game loader is also tasked with enforcing any usage conditions that might be associated with the game.

Looking at the attack scenarios described in the section "Gaming and piracy," we observe that sharing the installation media is discouraged in two ways. We assume that users are unwilling to share their Personal IDs (the registration flow links the Personal ID to the Media ID), but should they nevertheless decide to do so, the license server can limit the number of installations per user to a reasonable maximum. Furthermore, the drive authentication procedure requires the corresponding installation media to be physically present so that only one of the installed copies can be active at any given time. This architecture basically supports multiple installations with the same Media ID and Personal ID and facilitates multiple installed copies that are registered to the same user—either on the same machine or on different machines. The attacker could conceivably attempt to spoof the Media ID when registering the game, but the game installer can validate the Media ID by using the KMB and some content from the disk.

The installation medium itself is now subject to the copy-protection capabilities of CPPM. If the disk image is copied onto another blank medium, the game loader will not be able to calculate the correct title keys and will therefore fail to decrypt the game executables.

A more elaborate attack against which to defend is the removal of some or all of the protection layers in order to redistribute the game as an unprotected image. This can be done at two levels. Portions of the game executable will have to be in the clear as they are executed. If these portions are captured and reassembled together with the nonexecutable gaming content to form a complete, stand-alone distribution, the attacker still has to remove the fingerprint from the executable to prevent the pirate content from being traced back to him.

Alternatively, the attacker might try to obtain an unwatermarked copy of the game executable by extracting the device keys from the game loader, or by locating device keys on the Internet, and using them to obtain a decrypted copy of the game on the installation media. However, these game executables have been prepared for fingerprint insertion and are therefore not in an executable state. To succeed, the attacker has to go through the registration flow again and then successfully remove the digital fingerprint. Thus, our level of resilience against both of these attacks depends on the strength of our fingerprinting technology.

If a compromised game loader becomes publicly available on the Internet, new games can be released with updated KMBs that disable the hacked component. If the game loaders are using shared device keys, this revocation will trigger an update of all game loaders in the field.

We observe that by adding the registration protocol flows for the game executables, the role of copy protection for the installation media is diminished. In the next scenario, we present a model that completely abandons protection of the installation media and therefore enables electronic distribution and various forms of super-distribution in addition to the conventional distribution of games on media. Super-distribution, a common term used in discussing DRM systems, is a way of distributing freely and widely digital files that are protected by using tamper-resistant technologies to prevent modification and modes of usage not authorized by the vendor.

Electronic distribution of PC games

In this section we present a variation of the architecture described in the previous section. This model uses the copy-protection features of CPRM-protected recordable media to establish a strong tie between the game executables and the target machine. Also, with this model, we would like to enable secure online distribution of the game content. Thus, possession of the installation media is no longer necessary as a proof of ownership—the installation image can be obtained in many different ways. In this model, the user is charged and obtains a license for the game when the registration flow is completed.

As in the previous scenario, the model in *Figure 8* uses the game loader as an enabling tool that is protected by tamper-resistant software and contains device keys. However, in this case, the cryptographic keys to unlock the game executables are managed on a separate SD memory card, protected by the 4C copy-protection features described in the section "Content protection for physical media." The game executables themselves are encrypted and located on the PC. To obtain the decryption keys for running the game executables, the game loader has to be able to successfully process the KMB on the SD memory card by using its device keys. The built-in storage devices of the PC are used to extend the capacity of the SD memory card. In other words, the SD memory card is used to manage the inventory of licensed games for a given user.

Looking at the security properties of this model, we observe that protection of the installation media is no longer a concern. It does not matter if the installation image is copied, shared through a peer-to-peer network, or redistributed in other ways. As a matter of fact, with this model, the installation media itself could be given out without charge, for example, to people who subscribe to a video gaming journal. Only during the registration flow would the user actually acquire a license to the game. Therefore there is no need to be concerned about users sharing or copying the installation media. With respect to the other attacks discussed in the introduction, this model has the same characteristics as the architecture presented in the previous section.

Note that this model enables several interesting new features.

- The CPRM for Network Download architecture described in the section "Content protection for physical media" can be used in order to remotely associate new gaming content with the SD memory card. If combined with the registration flow for watermark insertion, this provides an effective method for electronic distribution.
- Even a small SD memory card (a 64-MB model currently retails for less than $20) can handle key material for a very large number of games. This avoids the check for the installation disk, which could be considered problematic from a usability perspective. A single SD memory card can easily be used to manage 1000 games and has a very high degree of portability.
- The secure state management capabilities of the SD memory card can be used to implement a variety of usage conditions to enable new business models. The scenarios that can be realized this way include a limited time rental, a "try before you buy" model that limits the number of plays, and others.
- Because the game registration flow includes the Media ID, a game can be installed safely on multiple machines with the same SD memory card. This enables support for multiple installations while only one of them can be active at a given time. The SD memory card literally becomes the key to unlock the installed game on the PC.

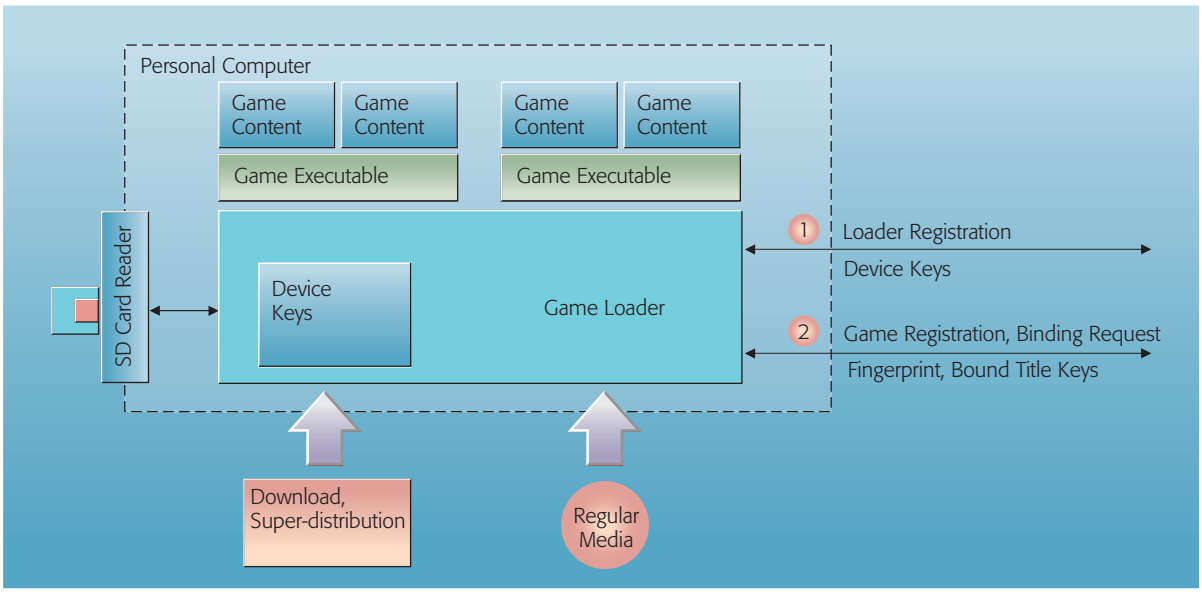

Figure 8
Electronic distribution of PC games

- Extending this model further, the SD memory card can also be used to store other game-related state information in unprotected mode, allowing the end user to consistently play a game across multiple installations on different machines by taking along his or her SD memory card.

To summarize, the significant change in this model is that the SD memory card replaces the installation media as a token of ownership. It no longer matters if the game was received on physical media or via download—the key to unlocking the installed game executable is managed on the SD memory card, taking advantage of the CPRM copy-protection features and providing a high degree of portability.

Management of virtual game assets with copy-protected media

Another requirement, which has surfaced with the increasing popularity of online games, is the need to secure virtual game assets that are of real value in the gaming community. These assets include not only certain attributes of a character in the game, which can only be acquired after playing for a long time, but also belongings of the character that are transferable. There is an active secondary market where these virtual assets are swapped or even bought and sold by users who play the game.

The need for protection of these virtual assets arises from their real world value. In certain scenarios, the need for protection can be addressed with copy-protection technology. In particular, this is the case if virtual assets are managed locally on the end user's machine and if these assets are transferable in a peer-to-peer fashion. The secure state management capabilities of copy-protected media can then be used to manage the inventory of virtual assets and to enable transfer operations, similar to the mechanisms used to handle copy or move operations with multimedia content.

The model presented in the previous section lends itself well to being extended to include this capability. *Figure 9* illustrates the extended model. In addition to storing the user-specific state information in unprotected mode on the SD memory card, any virtual assets can be stored in protected mode and will be subject to the copy-protection features of the CPRM-enabled media.

In this scenario, the SD memory card contains all the user-specific information, including the protected inventory of virtual assets. Device keys are needed to manage the inventory of assets on the SD memory card in protected mode and to perform a proper copy or move operation for transfer. Following the

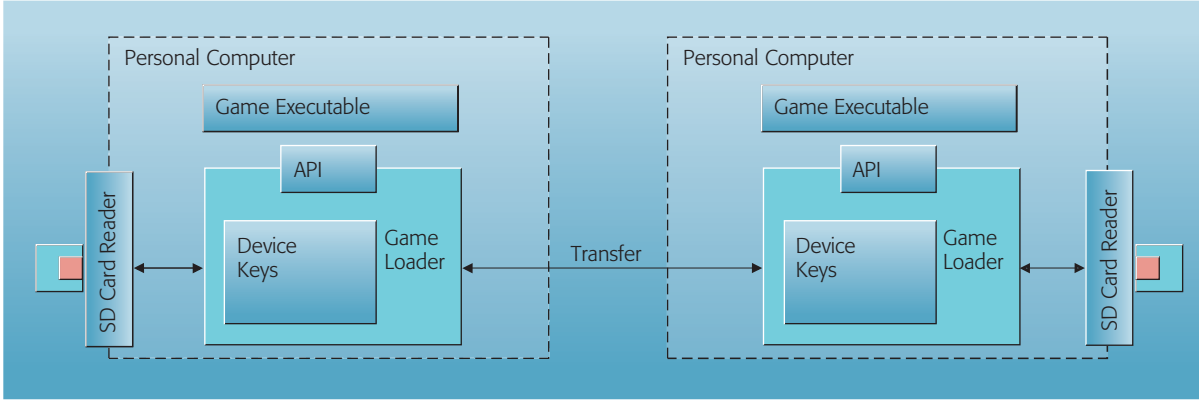

Figure 9
Management of virtual game assets on an SD Memory Card

architecture described in the previous section, the device keys are managed by the game loader. To make this asset management functionality available to the game executables, we propose to export this capability in the form of an application programming interface (label API), as shown in Figure 9. Note that some of these interfaces will require authentication of the calling code module with a code signature to make sure user assets are created and removed only by valid game executables.

In this model, the SD memory card serves as the storage container for all virtual assets created by a game. By leveraging the copy-protection features of the SD memory card, we provide a basic level of security for virtual assets and allow them to be managed locally, at the end user machine. This solution effectively prevents backup-restore type of attacks, such as a user selling an asset and then restoring a backup to recreate a copy of the asset.

A successful attack against this model will have to involve reverse engineering the game loader in order to obtain the device keys. In that case, the broadcast encryption-based revocation mechanism is our defense to enforce an update of the game loader applications.

CONCLUSION

In this paper we presented two novel approaches to copy protection for video games that target some of the shortcomings of current techniques. These techniques draw on previously developed protection technologies, which to the best of our knowledge have yet to be applied to video games. The first approach uses branch-based software watermarking to create game executables that are linked to the user. Through this technique the gaming industry can begin to address one of the most difficult aspects of video game piracy: identifying the source of an illegal distribution. As an alternative, we draw on the parallels between games and premium audio and video content to demonstrate how the broadcast encryption technology used by 4C can be applied to both PC and console-based games. The use of these open standards-based techniques would enable the gaming industry to move away from protection based on proprietary information and toward techniques that have a stronger scientific basis.

**Trademark, service mark, or registered trademark of Commodore-Amiga, Inc., Microsoft Corporation, Nintendo, Inc., Orion Pictures Corporation, SanDisk Corporation, Sony Computer Entertainment, Inc., Lenovo Ltd., Toshiba Corporation, or Valve Corporation in the United States, other countries, or both.

CITED REFERENCES

1. "Computer and Video Game Software Sales Reach Record $7.3 Billion in 2004," *Entertainment Software Association (ESA)* (Jan. 2005), http://www.theesa.com/archives/2005/02/computer_and_vi.php.
2. "2005 Special 301 Report on Global Copyright Protection and Enforcement," *International Intellectual Property Alliance (IIPA)* (Feb. 10, 2005), http://www.iipa.com/special301.html.
3. G. Myles and H. Jin, "Self-Validating Branch-Based Software Watermarking," in *Information Hiding, 7th*

International Workshop, Lecture Notes in Computer Science **3727**, Springer-Verlag Inc., New York (2005), pp. 342–356.

4. T. Maude and D. Maude, "Hardware Protection Against Software Piracy," *Communications of the ACM* **27** (9), 950–959, 1984.
5. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Bohen, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, New York (2000), pp. 168–177.
6. Trusted Computing Group Home, Trusted Computing Group, http://www.trustedcomputinggroup.org/home.
7. S. Parsons, "More on Xbox Live Bans," Joystiq, Weblogs Inc., (November 14, 2004), http://www.joystiq.com/entry/6673569691391113/.
8. F. B. Cohen, "Operating System Protection through Program Evolution," http://all.net/books/IP/evolve.html, 1992.
9. D. Libes, *Obfuscated C and Other Mysteries*, John Wiley and Sons, New York, 1993.
10. C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscation Transformations," Technical Report 148, Department of Computer Science, University of Auckland (July 1997).
11. C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *Proceedings of the ACM Symposium on Principles of Programming Languages,* ACM, New York (1998), pp. 184–196.
12. C. Collberg, C. Thomborson, and D. Low, "Breaking Abstractions and Unstructuring Data Structures," *IEEE International Conference on Computer Languages,* IEEE, New York (1998), pp. 28–38.
13. F. Hohl, "Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts," *Mobile Agents and Security, Lecture Notes in Computer Science* **1419**, Springer-Verlag Inc., New York (1998), pp. 92–113.
14. C. Wang, "A Security Architecture for Survivability Mechanisms," Ph.D. Thesis, University of Virginia, School of Engineering and Applied Science (Oct 2000).
15. C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Software Tamper Resistance: Obstructing Static Analysis of Programs," Technical Report CS-2000-12, University of Virginia, Dec. 2000.
16. C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Protection of Software-Based Survivability Mechanisms," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks,* IEEE Computer Society (2001), pp. 193–202.
17. C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security,* ACM, New York (2003), pp. 290–299.
18. D. Aucsmith, "Tamper Resistant Software; An Implementation," in *Information Hiding, First International Workshop, Lecture Notes in Computer Science* **1174**, Springer-Verlag Inc., New York (1996), pp. 317–333.
19. T. Sander and C. F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts," *Mobile Agents and Security, Lecture Notes in Computer Science* **1419**, Springer-Verlag Inc., New York (1998), pp. 44–60.
20. D. Aucsmith and G. Graunke, *Tamper Resistant Methods and Apparatus*, U.S. Patent 5,892,899, Assignee: Intel Corporation, 1999.
21. H. Chang and M. Atallah, "Protecting Software Code By Guards," in *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management, Lecture Notes in Computer Science* **2320**, Springer-Verlag Inc., New York (2001), pp. 160–171.
22. B. Horne, L. Matheson, C. Sheehan, and R. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance," in *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management, Lecture Notes in Computer Science* **2320**, Springer-Verlag Inc., New York (2001), pp. 141–159.
23. C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection," *IEEE Transactions on Software Engineering* **28**, No. 8, 735–746, August, 2002.
24. Steam, http://www.steampowered.com.
25. D. Grover, "*Program Identification,*" *The Protection of Computer Software: Its Technology and Applications*, The British Computer Society Monographs in Informatics, Cambridge University Press, Second Edition, 1992.
26. K. Holmes, *Computer Software Protection*, U.S. Patent 5,287,407, Assignee: International Business Machines Corporation, Feb. 1994.
27. P. R. Samson, *Apparatus and Method for Serializing and Validating Copies of Computer Software*, U.S. Patent 5,287,408, Assignee: Autodesk, Inc., Feb. 1994.
28. S. A. Moszowitz and M. Cooperman, *Method for Stega-Cipher Protection of Computer Code*, U.S. Patent 5,745,569, Assignee: The Dice Company, Jan. 1996.
29. R. L. Davidson and N. Myhrvold, *Method and System for Generating and Auditing a Signature for a Computer Program*, U.S. Patent 5,559,884, Assignee: Microsoft Corporation, Sept 1996.
30. C. Collberg and C. Thomborson, "Software Watermarking: Models and Dynamic Embeddings," in *Proceedings of the Symposium on Principles of Programming Languages,* 1999, pp. 311–324.
31. G. Qu and M. Potkonjak, "Hiding Signatures in Graph Coloring Solutions," in *Information Hiding, Third International Workshop, Lecture Notes in Computer Science* **1768**, Springer-Verlag Inc., New York (1999), pp. 348–367.
32. J. P. Stern, G. Hachez, F. Koeune, and J. J. Quisquater, "Robust Object Watermarking: Application to Code," *Information Hiding, 3rd International Workshop, Lecture Notes in Computer Science* **1768**, Springer-Verlag Inc., New York (1999), pp. 368–378.
33. A. Monden, H. Iida, K. Matsumoto, K. Inoue, and K. Torii, "A Practical Method for Watermarking Java Programs," in *Proceedings of the 24th Computer Science and Applications Conference,* IEEE Computer Society, 2000, pp. 191–197.
34. R. Venkatesan, V. Vazirani, and S. Sinha, "A Graph Theoretic Approach to Software Watermarking," *Information Hiding, 4th International Workshop, Lecture Notes in Computer Science* **2127**, Springer-Verlag Inc., New York (2001), pp. 157–168.
35. Genevieve Arboit, "A Method for Watermarking Java Programs via Opaque Predicates," in *Proceedings of the Fifth International Conference on Electronic Commerce Research* (ICECR-5), 2002.

36. P. Cousot and R. Cousot, "An Abstract Interpretation-Based Framework for Software Watermarking," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2004), pp. 173–185.
37. C. Collberg, E. Carter, S. Debry, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp, "Dynamic Path-Based Software Watermarking," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (2004),* pp. 107–118.
38. J. Nagra and C. Thomborson, "Threading Software Watermarks," *Information Hiding, 6th International Workshop, Lecture Notes in Computer Science* **3200**, Springer-Verlag Inc., New York (2004), pp. 208–233.
39. J. Nagra, C. Thomborson, and C. Collberg, "A Functional Taxonomy for Software Watermarking," in *Proceedings of the Twenty-Fifth Australasian Computer Science Conference,* Australian Computer Society Inc. (2002), pp. 177–186.
40. J. M. Nardone, R. P. Mangold, J. L. Pfotenhauer, K. L. Shippy, D. W. Aucsmith, R. L. Maliszewski, and G. L. Graunke, *Tamper Resistant Methods and Apparatus*, U.S. Patent 6,205,550, Assignee: Intel Corporation, March 20, 2001.
41. M. L. Fredman, J. Komlos, and E. Szemeredi, "Storing a Sparse Table with O(1) Worst Case Access Time," *Journal of the ACM* **31**, No. 3, 538–544 (July 1984).
42. K. Mehlhorn and A. K. Tsakalidis, "Data Structures," in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, J. van Leeuwen, Editor, pp. 301–341, MIT Press, Cambridge, MA (1990).
43. SPEC CPU2000, http://www.spec.org/cpu2000/, 2001.
44. A. Fiat and M. Naor, "Broadcast Encryption," in *Advances in Cryptology (Crypto 93), Lecture Notes in Computer Science* **773**, Springer-Verlag Inc., New York (1994), pp. 480–491.
45. D. M. Wallner, E. J. Harder, and R. C. Agee, "Key Management for Multicast: Issues and Architectures," RFC 2627 (informational), The Internet Society (July 1999).
46. C. K. Wong, M. Gouda, and S. Lam, "Secure Group Communications Using Key Graphs," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication,* ACM, New York (1998), pp. 68–79.
47. D. Naor, M. Naor, and J. Lotspiech, "Revocation and Tracing Routines for Stateless Receivers," *Advances in Cryptology (Crypto 2001), Lecture Notes in Computer Science* **2139**, Springer-Verlag Inc., New York (2001), pp. 41–62.
48. Publications and Current Versions, 4C Entity, Intel Corporation, IBM, Matsushita Electric Industrial Co., Toshiba Corporation, http://www.4centity.com/docs/versions.html.
49. For additional background, see C. Brendan and B. Traw, "Protecting Digital Content Within the Home," *Computer* **34**, No. 10, 42–47 (Oct 2001).
50. "Content Protection for Recordable Media Specification—Network Download Book," Intel, IBM, MEI, Toshiba (2004), http://www.4centity.com/licensing/adopter/CPRM-Download-090.pdf.

Accepted for publication August 30, 2005.
Published online January 18, 2006.

Ginger Myles
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 (gmyles@us.ibm.com). Ginger Myles is currently a Postdoctoral Scientist at IBM's Almaden Research Center and is finishing a Ph.D. degree in computer science at the University of Arizona. She received a B.A. degree in mathematics from Beloit College in Beloit, Wisconsin and an M.S. degree in computer science from the University of Arizona. Her research focuses on all aspects of content protection.

Stefan Nusser
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 (nusser@us.ibm.com). Dr. Stefan Nusser is a research staff member at IBM's Almaden Research Center and manages a research team focused on content protection. His research interests include content protection and digital rights management. He received a Ph.D. degree in management information systems from Vienna University of Business Administration and Economics in Vienna, Austria. ■