Strategies to Combat Software Piracy

JAYADEV MISRA*
The University of Texas at Austin
Austin, Texas 78712
Email: misra@cs.utexas.edu
Home page: http://www.cs.utexas.edu/users/misra

August 10, 1999

Abstract

It is impossible to combat software piracy as long as the machines on which the programs execute are indistinguishable; then, any program that can execute on one machine may be copied for execution on another machine. Recently, hardware manufacturers have begun assigning unique identifiers to CPU chips, which make it possible to address the piracy issue in a new light. In this paper, we suggest schemes that make it nearly impossible to use certain kinds of software on a machine unless the manufacturer of that software has issued a license for that specific machine.

1 Introduction

It is impossible to combat software piracy as long as the machines on which the programs execute are indistinguishable; then, any program that can execute on one machine may be copied for execution on another machine. Recently, hardware manufacturers have begun assigning unique identifiers to CPU chips, which make it possible to address the piracy issue in a new light. In this paper, we suggest schemes that make it nearly impossible to use certain kinds of software on a machine unless the manufacturer of that software has issued a license for that specific machine.

Terminology An *author* creates *files*; an author is, typically, a software manufacturer, and a file consists of executable as well as data components (music and video files as well as binary executables can be protected using our scheme). A *retailer* or the author sells a copy of the file to a *customer*. The customer

*This material is based in part upon work supported by the National Science Foundation Award CCR–9803842.

installs the file on a *machine* and the file *opens* on that machine under a *kernel*. Additionally, a group of individuals may create and distribute files among themselves; we call such files *privately shared files*. □

From the viewpoint of the author neither the retailer nor the customer is trustworthy. A retailer may make unauthorized copies of a file and sell them to customers at lower prices, and a customer may let other customers "borrow" the files he has bought for a small fee. Additionally, a fake author may alter the code of a genuine file, claim that the altered file is his creation and market the file under a different title, possibly, through a dishonest retailer. Our scheme prevents piracy in spite of such collusion among the dishonest parties.

Which files can be protected? The piracy problem, in theory, is unsolvable. Consider, for instance, a file whose content is a genome sequence. The distributor of the file must provide the means for a customer to extract the symbol at every position in the sequence. Therefore, a customer can recreate the sequence, encode it in a different format and market the file. Similarly, a file containing a video stream can be replicated by simply playing the video. Call a file f *vulnerable* if a file equivalent to f can be constructed easily given the specification of f and the outcomes of a finite number of experiments with f. The genome sequence and the video stream are vulnerable, as we have argued. By contrast, a word processing program is not easily reconstructed from its specification (i.e., its manual) given a reasonable amount of time to experiment with it. Methods of water-marking can be used to protect music and video files – see section 3.3 for certifications of such files by a trusted third party – and genome sequences are best protected manually through patenting.

Our thesis is that a file can be protected against piracy, without employing a trusted third party, if and only if it is not vulnerable. This statement, unfortunately, carries little formal meaning since neither of the important terms is precisely defined. Yet, we use this thesis as a guiding principle in devising antipiracy schemes.

Requirements for an Antipiracy Scheme Our requirements for antipiracy are grouped into correctness and performance requirements. Correctness requirements are stated more precisely than the performance requirements, because the latter may be implemented in a variety of different ways by the authors and the kernels. Most of the following requirements are self-explanatory.

- Correctness Requirements:

    1. A file opens on a machine only if the author of that file has authorized the machine for opening that file.

- Performance Requirements:

    1. The author does not have to make extraordinary efforts to build pirate-proof files. Few burdens are imposed upon the customer in installing and opening the file, and the retailer in selling it.

2. Storage and processing requirements for opening a file at the kernel level are minimal.

Assumptions Our scheme requires the kernel to verify a "license"; so, we require a secure kernel. Additionally, as mentioned earlier, each machine has a unique id that can not be compromised; effectively, the machine id is part of the kernel.

Note that having a secure kernel alone does not solve the piracy problem because the authenticity information from a genuine file (purchased for one machine) may be attached to a pirated file running on the same machine. Our solution makes heavy use of public key cryptography [2], – in particular, signed messages – which we assume to be secure.

1. Each machine operates under a *secure kernel*. That is, the code of the kernel cannot be changed, its state can not be examined and its accesses to addresses (in the application programs) can not be observed.

2. Each machine has a unique identifier, called its *machine id*, that cannot be changed.

3. Public key cryptography is secure.

Overview of the Solution and the Paper In order to buy a file a customer provides his machine id. The customer receives a file along with a *license*: the license contains the file id and machine id, and the license is signed by the author. The kernel allows execution of a file on a specific machine only if the file is licensed to run on that machine.

Under mild attack, see section 3, a pirate can only copy a file, verbatim. This is currently the prevalent mode of attack, and this can be combatted relatively easily by verifying the license.

We also consider *vicious attacks* in section 3, in which a pirate may read, analyze and modify a file. In particular, the pirate may change the file header so that a license for another product can be used for this file. We propose a few schemes that thwart this attack. Methods to combat dishonest retailers are discussed in section 4. Several aspects of our scheme, including a few extensions, are discussed in section 5. Since many ingenious attacks may be imagined, we construct a formal proof of the main result based on a small number of axioms that couch our assumptions. The only attacks that can succeed against our scheme are the ones that invalidate the axioms; therefore, we discuss each axiom in detail and the possible attacks against it.

Related Work There is a vast amount of work on secure transactions. We mention only two here: digiboxes from Intertrust Corporation [4] and cryptolopes from IBM [1]. These employ methods similar to block encryption that we suggest in section 3.2.

2 Axiomatic Treatment of Piracy and Licensing

2.1 File Header

Each author and each kernel has a unique *signature*. A signature is a function that converts a string to another string[1]. Any one can decrypt a string signed by author v, but no one other than v can create a string with v's signature. We write $v.sign, v.unsign$ for the signature encryption and decryption functions for v; $v.sign$ is private to v, but $v.unsign$ is publicly available. We assume that $v.sign$ and $v.unsign$ are inverses of each other.

Each file, f, has a header, $f.header$, that includes

$f.id$, the title of file f, and

$f.unsign$, the (publicly available) signature decryption function of the author of f.

The header information in a file may not be genuine because it may have been corrupted by the pirate.

Privately Shared Files We regard each customer as an author. For a file created by a customer, the header information will, typically, be constructed by the kernel (it will assign an id to the file and the signature function may be picked from a table of such functions built into the kernel). This permits treatment of privately shared files in exactly the same way as the files marketed as products.

2.2 A Model of Piracy

In the world of piracy, a file is either *genuine* or a copy of a genuine file. A copy need not be verbatim; a pirate may analyze and simulate a genuine file to understand its functioning and create a file different from the original. But we expect that a copy would be very similar to the genuine file. Also, we expect genuine files from different authors to be quite different. Finally, we propose that genuine files from the same author should be made quite different: files with different version numbers, for instance, can be made dissimilar by loading the modules in a totally different order. Therefore, the only way two files are alike is for one to be a copy of the other (or even be a copy of a copy), perhaps with a small amount of modification.

Since files are either quite alike or quite different, we can define an equivalence relation, *similar* (written as $\sim$), over files. If files f, g are copies (or copies of copies) of the same genuine file then we expect $f \sim g$.

Axiom 1 asserts that each file is a copy of some unique genuine file. Axiom 2 says that a copy carries the same header information as the original. Axiom 2 seems hard to justify since a pirate can read a file, locate the header and replace it by a different header. We suggest methods that defeat this attack; see section

[1] A tuple of strings may be signed, by converting the tuple to a simple string. This conversion is not shown explicitly.

3.

Axiom 1: There is an equivalence relation, $\sim$, over files such that for any file f there is a unique genuine file g where $f \sim g$.

Axiom 2: $(f \sim g) \wedge (\text{genuine } g) \Rightarrow (f.header = g.header)$.

Remarks on the axioms It may seem unreasonable to claim that a copied file that is several generations removed from the original is still similar to the original. We do not believe that copies will be made from copies unless the copies are verbatim; therefore, this is a reasonable assumption.

Axiom 2 permits two unrelated files to have the same header. For instance, an author may release a file g that has the same header as f released earlier by the same author. This permits every purchaser of file f to upgrade to g at no additional cost; see section 2.3.

2.3 Licensing

When a customer buys a file f (from an author or retailer), he specifies his machine id, m. The customer receives f and a license $(f.id, m)$ signed by the author of f. A license can not be forged since it is signed. Privately shared files are acquired in exactly the same fashion, though the details will differ on the financial aspects of the transaction.

Opening a File A file can be opened on a machine only if the customer can produce an appropriate license. Specifically, the kernel on machine m carries out a license verification in order to open a file f under license c, as follows. It reads the header information $(f.id, f.unsign)$ from the file. (We describe in section 3 how the header information is encoded in a file so that it can not be removed by the pirate and it can be accessed by the kernel.) The kernel then computes $f.unsign(c)$; if this matches $(f.id, m)$ then the file is opened, else the file is deemed to be a fake.

Definition: m opens $(f, c) \equiv [f.unsign(c) = (f.id, m)]$.

It is reasonable to say that a machine m has been authorized to open file g if there is a license c such that m opens (g, c). Since no machine is authorized to open a non-genuine file, we may also assume that g is genuine if m has been authorized to open g.

Definition: m authorized for $g \equiv [(\exists\ c :: m \text{ opens } (g, c))\ \wedge\ \text{genuine } g]$.

2.4 Correctness

The correctness requirements are described in section 1. Our proof obligation is that if a machine opens file f under license c, i.e., m opens (f, c), then m has

been authorized for f. This result is not true because f may not be genuine. We can show, however, that there is a genuine g such that $f \sim g$ and "m authorized for g". Then, it makes no sense to open copy f on m given that permission to open the genuine file g on m has has been acquired.

Observation 1: Given f and m there is a unique c such that m opens (f, c). That is, m opens (f, c) and m opens (f, c') implies $c = c'$.

Proof: We prove the result by computing the value of c given that m opens (f, c).

$$
\begin{array}{ll}
 & m \text{ opens } (f, c) \\
\Rightarrow & \{\text{Definition}\} \\
 & f.unsign(c) = (f.id, m) \\
\Rightarrow & \{\text{Function application}\} \\
 & f.sign(f.unsign(c)) = f.sign(f.id, m) \\
\Rightarrow & \{f.sign, f.unsign \text{ are inverses}\} \\
 & c = f.sign(f.id, m)
\end{array}
$$

Note: It is highly improbable, though not impossible, for m to open two different files f, g using the same license, c. From observation 1, then
$c = f.sign(f.id, m)$, and
$c = g.sign(g.id, m)$.

This is a remote possibility for well-chosen signature functions, and even if these equalities hold for some m they are unlikely to hold for many other machine ids; thus, systematic piracy is still highly unlikely.

Theorem 1: m opens $(f, c) \Rightarrow (\exists\ g :: f \sim g \wedge \text{genuine } g \wedge m \text{ authorized for } g)$.

$$
\begin{array}{ll}
 & m \text{ opens } (f, c) \\
\Rightarrow & \{\text{Axiom 1}\} \\
 & (\exists\ g :: f \sim g\ \wedge\ \text{genuine } g)\ \wedge\ m \text{ opens } (f, c) \\
\Rightarrow & \{\text{Axiom 2 applied to } f \sim g\} \\
 & (\exists\ g :: f \sim g\ \wedge\ \text{genuine } g\ \wedge\ f.header = g.header) \\
 & \wedge\ m \text{ opens } (f, c) \\
\Rightarrow & \{\text{Definition of } header \text{ and "opens"}\} \\
 & (\exists\ g :: f \sim g\ \wedge\ \text{genuine } g \\
 & \ \wedge\ f.id, f.unsign = g.id, g.unsign \\
 &) \\
 & \ \wedge\ f.unsign(c) = (f.id, m) \\
\Rightarrow & \{\text{Predicate Calculus}\} \\
 & (\exists\ g :: f \sim g\ \wedge\ \text{genuine } g\ \wedge\ g.unsign(c) = (g.id, m)) \\
\Rightarrow & \{\text{Definition of "opens"}\} \\
 & (\exists\ g :: f \sim g\ \wedge\ \text{genuine } g\ \wedge\ m \text{ opens } (g, c)) \\
\Rightarrow & \{\text{Definition of "authorized"}\} \\
 & (\exists\ g :: f \sim g\ \wedge\ \text{genuine } g\ \wedge\ m \text{ authorized for } g)
\end{array}
$$

3 Attacks

The only assumptions we have made so far are, in rough terms: (1) the kernel is secure, (2) the machine id can not be compromised, (3) (axiom 1) each file is a copy of a unique genuine file, and (4) (axiom 2) the header of a genuine file can not be altered. We do not discuss the possible attacks against the kernel in this paper. The remaining assumption that needs scrutiny is axiom 2. We discuss two different kinds of attacks and show how this axiom can be satisfied.

Mild Attack In this form of attack, a pirate can only copy a genuine file verbatim. That is,

$(f \sim g) \equiv (f = g)$.

Then, a pirate can not change the header of a genuine file, and axiom 2 holds. More formally,

Observation 2: In mild attack, $(f \sim g) \Rightarrow (f.header = g.header)$.

Proof:

$$
\begin{array}{ll}
 & f \sim g \\
\Rightarrow & \{\text{Definition of mild attack}\} \\
 & f = g \\
\Rightarrow & \{\text{Predicate calculus}\} \\
 & f.header = g.header
\end{array}
$$

Mild attacks can be defeated even in the absence of a secure kernel; file f mimics the steps of the kernel for license verification. There is an executable portion in f that reads the header information from f, the license, c, the machine id m, and it evaluates (m opens (f, c)).

Vicious Attack In our treatment of vicious attacks, we endow the attacker with extraordinary powers. An attacker may replace portions of a binary file, simulate executions of portions of the file, take core dumps after execution of each instruction and analyze those to pinpoint instructions that change contents of specific memory locations, for instance. Almost all of these attacks are infeasible. For instance, replacing a portion of a binary executable file may cause all the absolute addresses to be shifted, which would make the execution of a program impossible. Yet, we allow these attacks in order to study the ultimate defenses.

There is no guaranteed way of preventing a pirate from reading (the binary contents) of a file, analyzing the contents to remove the header, and replacing it by a different header. A customer who plans to steal the file for personal use will replace the header by the header of a file that is licensed to run on his machine. But for large scale operation, a pirate declares himself to be an author and includes a header in a file identifying him as the author (replacing the original one). From then on, he is free to issue licenses for the file on any machine. It may seem that this particular attack can be thwarted by hiding

the header inside the file in some fashion so that it can not be removed; the exact hiding place is known only to the author and the kernel. But, then, any dishonest author can glean this information by claiming that he needs to hide the header in the files he produces. Therefore, the exact hiding place has to be author-dependent. This requires each kernel to maintain a list of authors and their hiding places; a unworkable solution if the author list is long and constantly changing.

A possible solution to this problem is for the file itself to do the license verification, as was proposed for the mild attack. The header and the code for the check could be hidden within the file. This is surely quite effective in practice. But, it is not immune to the kind of attack we have described earlier. A pirate can analyze the code to locate all instructions that compare a value against the machine id, for instance. He may then replace those instructions by the ones that compare the value against a location where a fake machine id is stored. While such attacks can be combatted by a variety of means to hide the code for license verification – indirect addressing, self-modifying code – there seems to be no general cure against a committed adversary who is allowed to analyze the code and simulate its execution. We suggest several possible defenses, each of which ensure axiom 2, that may be appropriate under different circumstances.

3.1 File Encryption

The simplest strategy is to encrypt the entire file so that only the kernel can decrypt it. The header information is a part of the file, and it will be impossible to remove the header from the encrypted file. The kernel opens the file by decrypting it and checking the header information. The file is executed in a privileged mode under the control of the kernel so that no attacker can read the decrypted version. This strategy can also be applied if the hardware supports execution of instructions in encrypted mode.

The major drawback of this strategy is the performance penalty associated with decryption each time the file is opened. Therefore, this strategy is best applied for small or infrequently used files.

3.2 Block Encryption

This is a refinement of the previous strategy; instead of encrypting the entire file, only a portion of it that includes the header is encrypted. We postulate that every file, f, that is not vulnerable, contains a block of code, B, with the following properties.

- B is *essential*: Unless B is executed at the appropriate point the machine will eventually crash or produce meaningless results.
- B is *incomprehensible*: Observing only the effects of B, i.e., which bits are altered as a result of executing B, for any finite number of inputs gives no clue about its effect for some other input.

The availability of such a block of code within f makes it possible to satisfy axiom 2, as follows. Include the header within B. Next, encrypt B so that it can be decrypted only by the kernel. The encrypted version of B, B', is passed as an argument of a call to a kernel routine. The kernel routine decrypts B', checks for authenticity using the header included there and then executes the code of B. We argue that this strategy satisfies axiom 2.

- B' can not be removed from a copy of f because B is essential.
- An attack may attempt to replace B' by B'' where B'' is (the encrypted version of) some code that mimics B and B'' includes a fake header. Since B' can not be analyzed by an attacker, being in an encrypted form, the construction of B'' has to rely on the observations of the effects of B', i.e, observing the bits that are altered as a result of executing B'. (The storage area for the kernel is unobservable, and, hence, the decrypted version of B' is also unobservable.) From the condition of incomprehensibility, no finite set of experiments suffice to determine the functionality of B'. Hence, no such B'' can be constructed.
- From the two observations above, every copy of f has to include B'. Therefore, the header is included, as required by axiom 2.

The block B is application dependent. For instance, in a program that performs garbage collection, a few critical instructions manipulate the pointers. Understanding the effects of these instructions is tantamount to understanding the full program. These critical instructions can be taken to be B. However, in a larger program that includes a garbage collector, it is not sufficient to use this strategy, because an attacker may replace the entire garbage collector.

We have assumed that B is executed only once. We can make it extremely difficult for the attacker to observe the effect of B if B spawns a kernel process that runs concurrently with the application program. This process does the license verification and carries out some essential computation. Observing the effect of B is very difficult since its execution is intertwined with the execution of the other processes in f and its effect is spread out over an execution of undetermined length.

3.3 File Certification

An entirely different scheme to thwart vicious attacks is based on using unforgeable *certificates*. There is a trusted third party, called a certification authority, to whom authors submit their files for certification. If the authority determines that the file is genuine and the author is the legitimate creator of the file then it issues a certificate to the file, which the author attaches to the file as a header. The kernel checks for such a certificate in each file and uses the certificate and the license to open a file.

The certification authority may use a variety of means to check the authenticity of files. For music and video, water-marking may be the preferred solution;

for a genome sequence a manual patent check may be sufficient. We outline an automatic scheme, similar to water-marking, that is applicable to executable binary files. Before explaining the scheme, we state the properties of certificates in axiom $3-5$, and show that axiom 2 is implemented. Then, we show that our scheme implements these axioms. Note that any scheme that satisfies these axioms will protect the files against piracy; our scheme is only one way of implementing the axioms.

A certificate is issued only to a genuine file and for a specific kernel (imagine that files running under different kernels are different). For every file f, $f.header$ is a certificate. For a genuine file, g, $g.header$ is the certificate issued to it, and for a fake file, f, $f.header$ is a certificate issued to some genuine file. A kernel can verify if a file "satisfies" a certificate; we write $f\ sat\ c$ to denote that file f satisfies certificate c, and we define this term in section 3.3.1. Note that f need not be genuine to satisfy c.

Axiom 3: For any file f, $f\ sat\ f.header$.

Axiom 4: For any certificate c, $f\ sat\ c\ \wedge\ g\ sat\ c \Rightarrow f \sim g$.

Axiom 5: For every certificate, c, there is a genuine file f such that $c = f.header$.

We now prove that axiom 2 follows from the three axioms above.

Theorem 2: $(f \sim g) \wedge (\text{genuine } g) \Rightarrow (f.header = g.header)$.

Proof: Let $c = f.header$.

$$
\begin{array}{ll}
 & c = f.header \\
\Rightarrow & \{\text{Axiom 5: let } h \text{ be a genuine file such that } h.header = c\} \\
 & c = f.header \wedge c = h.header \wedge (\text{genuine } h) \\
\Rightarrow & \{\text{Axiom 3: Both } f, h \text{ satisfy their headers}\} \\
 & f\ sat\ c \wedge h\ sat\ c \wedge f.header = h.header \wedge (\text{genuine } h) \\
\Rightarrow & \{\text{Axiom 4: } f\ sat\ c \wedge h\ sat\ c \Rightarrow f \sim h\} \\
 & f \sim h \wedge f.header = h.header \wedge (\text{genuine } h) \\
\Rightarrow & \{\text{antecedent: } f \sim g \wedge (\text{genuine } g)\} \\
 & f \sim g \wedge (\text{genuine } g) \wedge f \sim h \wedge (\text{genuine } h) \wedge f.header = h.header \\
\Rightarrow & \{\text{Uniqueness condition from axiom 1}\} \\
 & g = h \wedge f.header = h.header \\
\Rightarrow & \{\text{Predicate calculus}\} \\
 & g.header = h.header \wedge f.header = h.header \\
\Rightarrow & \{\text{Predicate calculus}\} \\
 & f.header = g.header
\end{array}
$$

3.3.1 Implementing Axioms 3,4

We propose that a certificate issued to f contain a pair $(f.id, f.sign)$, matching the structure of a header; the certificate will be signed by a trusted third party, as we show in section 3.3.2. The component $f.id$ of the certificate is computed as follows. Let p be a sequence, $p = p_0, p_1, ..,$ where, for all i, p_i is a random position in f where the bit value is 0. Then $f.id$ is $K.unsign(p)$ where K is the kernel under which this file operates (recall that each kernel has a signature). Thus, $f.id$ can be read (i.e., decrypted) only by the kernel.

Definition: File f satisfies certificate c (written as f *sat* c), where c contains the pair (u, v), if $f[p_i] = 0$, for all i, and $p = K.sign(u)$.

We claim that axioms 3,4 are met by this scheme. For a genuine file, g, the certificate issued to it becomes its header; therefore, g *sat* $g.header$, meeting axiom 3. For any other file the kernel rejects the file if this condition is not met. Therefore, every file submitted to a kernel meets axiom 3. For axiom 4, note that a random bit string satisfies c with probability $2^{-|p|}$, where $|p|$ is the length of the sequence of positions chosen from f. Thus, for sufficiently long p, say $|p| = 40$, it is highly unlikely for two files to satisfy the same certificate unless they are copies. Additionally, no one can tweak a file content to satisfy a certificate c for an entirely different file, f, because that would require decryption of $f.id$; only the kernel can perform this decryption.

3.3.2 Implementing Axiom 5

We use a trusted third party, called a Certification Authority, or CA for short, for implementing axiom 5. There can be several CA. An author submits to a CA a file f and the name of the kernel K under which f is to be executed. If the CA determines that f is genuine then it issues a signed certificate, $CA.sign(f.id, f.sign)$, that becomes $f.header$. Since the certificate is signed by a CA, any fake certificate will be rejected by the kernel. Therefore, axiom 5 is met: For every certificate, c, there is a genuine file f such that $c = f.header$.

The remaining question is how a CA tells if a file is genuine. Here, different methods may be employed for different kinds of files. Water-marking is best applied for music and videos; if the CA detects a water-mark of another author it rejects the file. We describe below a variation of this scheme to establish authenticity of executable binary files.

For CA to issue certificates only to genuine files the author has to prove to CA that a file it submits is genuine. One possible proof is for the author to supply the source code for f; any party that has access to the source code can be deemed to be the genuine author. CA then compiles the source code and certifies the resulting executable file. Unfortunately, this scheme does not quite work, because it requires the author and CA to agree on a common compiler. It is possible that an author has an in-house compiler on which the code is to be compiled. Then, CA has to first certify the genuineness of the compiler. This is

not a frivolous issue, because a pirate may create a fake compiler that includes an encoding of a genuine executable file, f, from another author; compiling any input with this compiler produces f. If CA accepts compilers without question from authors then it will certify f as a creation of the pirate.

We suggest a strategy based on water-marking. Each author first registers with CA and then CA assigns an *imprint* to the author. The imprint information is a secret that is shared between CA and the author. Imprints have the following properties: (1) it is not burdensome for an author to embed its imprint into its files, (2) it is easy for CA to check a file for the imprints of all registered authors, (3) it is highly unlikely that a random bit string bears the imprint of any author, and (4) it is nearly impossible to remove an imprint from a file without knowing the imprint. We discuss certain kinds of imprints and argue why they do or do not meet these conditions.

First, we show that CA can certify that a genuine file is genuine and reject fake files, given these properties of imprints. For a file submitted by author v, CA checks that the file bears v's imprint and only v's imprint. Then CA can assert that the file is written by v; had it been written by any other author it would bear that author's imprint, from condition (4).

A genuine file submitted for certification may bear another author's imprint accidentally. In that case, the file will be rejected by CA, and the author will have to modify it. The chances of this happening are quite low (see assumption 3 above), and it is highly unlikely that this scenario will repeat with the modified file.

How can an attacker change an executable file? We have allowed the possibility that an attacker may change a genuine file, f, to a file g and then market g as his own. What kinds of changes are possible?

A genuine executable file, f, is available as a bit string. Let $f[i]$ denote the bit value at the i^{th} position of f. Since the source code of f is not available, the attacker has, at best, an understanding of small portions of the code of f. Without a thorough understanding of the file it is impossible to change it extensively and still have the file run properly. Thus, inserting an instruction, which shifts a significant portion of the memory map by one location, is surely going to make an executable file useless. Similarly, data items cannot be changed significantly either because their absolute addresses may be used in the file. The pirate may insert new code at the end of f and change small portions of f by adding jumps to new code. He may also modify a few bits of f (even at the expense of losing some functionalities) in order to embed his imprint. However, for $f \sim g$, we can assert that $f[i] = g[i]$, for almost all i whenever $f[i], g[i]$ are defined. If this is not the case then g will not execute because file f makes references to absolute addresses in f, which will now be different in g. We exploit this limitation of the attacker to distinguish a genuine file from a fake file.

Notation: *position* is an index i such that $f[i]$ is defined. For a sequence of positions X, $f[X]$ is the bit string obtained from the corresponding positions of

f; we assume that all positions in X fall within the bounds of f so that $f[X]$ is defined.

Imprints that meet the stated conditions A simple imprint that CA can assign to each author is a pair (X, S), where X is a sequence of positions and S is a bit string, and X and S have the same length. A bit string f bears the imprint (X, S) if $f[X] = S$. The probability that a random string bears this imprint is $2^{-|X|}$. However, it is possible to guess such an imprint by examining a large number of files from the same author. Suppose we examine q files from an author and find that all files have the same value at a particular position. The likelihood of this occurrence is 2^{-q+1}; for q of about 50, say, it is extremely unlikely for one position to have the same value in all files unless it is specifically intended. Thus, all such positions can be identified and they can all be inverted; with high likelihood, the imprint will then be removed.

This analysis assumes that different files from the same author share nothing but the imprint. This may be true for music and video files, but for executable files they, typically, share much more. So, it is unlikely that this attack will succeed in practice. However, we can modify this imprinting scheme and avoid the attack altogether.

As before, let CA assign to each author a pair (X, S), where X is a sequence of positions and S is a bit string, and X and S have the same length. A bit string f bears this imprint if $f[X] = S'$, where S' is a rotation of S by some amount. We argue below that this scheme meets all the criteria for imprints listed earlier. In the following, let N be the number of bits in f (N is several hundred million for today's commercially available files) and n be the length of X (and S); (we will consider n in the range of about 32 bits).

First, It is possible for an author to embed its imprint into its files if n is small. For $n = 32$, for instance, it will take some help from compiler and loader to put a few hand-designed routines into specific positions in f so that the imprint is embedded. Or, it may be possible to fiddle with absolute code to invert a few bits to embed an imprint.

Second, CA can check a file for a specific imprint quite easily: given (X, S) to see if $f[X]$ is a rotation of S can be solved using an algorithm due to Shiloach [3] that operates in time proportional to $|X|$; see Gasteren and Feijen [5] for an elegant development of Shiloach's algorithm. More simply, it is sufficient to check that $f[X]$ is a substring of SS, and a linear string matching algorithm can be used for this purpose. The imprints of all authors in a given file can be checked in time proportional to the number of authors and the lengths of their imprints. (For $30,000$ authors it takes around 3 seconds to check all the imprints, assuming that 10^7 instructions are executed per second, and that it takes around 1000 instructions to check for an imprint.)

Third, it is quite unlikely that a random file bears the imprint of some author. The probability that a random bit string has the imprint (X, S) of a specific author is $n/2^n$ (we assume that the string S yields different strings when rotated by different amounts). Given that there are r authors, each with

the same parameter n, the probability of finding the imprint of some author in a random string is, $1-(1-n/2^n)^r$. For $n, r = 32, 30000$, this is about 0.0002. This is the probability that CA rejects a valid file from an author because it detects the imprint of another author. The author may recompile the file in a different sequence and submit it for certification in that case.

Fourth, we argue that it is nearly impossible to remove an author's imprint from a file without knowing the imprint itself. An imprint can be removed from f by creating g which is a shift of f by one position. It is unlikely then that $g[X] = f[X]$, and hence $g[X]$ will most likely differ from any rotation of S, thus removing any imprint in f. However, g will then differ from f in a large number of positions (around half, on average), and, as we have argued, g will not then execute properly. As long as most positions in g have identical bit values as in f, the only attack seems to be to guess some position in X and invert the bit value at that position. The only information available to an attacker are several files from the same author. However, no position in all the files bears any distinguishing characteristic if S is chosen appropriately, for instance, if S has about equal number of zeroes and ones. Then, the only attack that seems to be left is to invert a random number of bits of f to remove an imprint.

How many bits have to be inverted randomly so that an imprint is removed? We calculate the probability of choosing one of the positions in X if t positions are chosen randomly out of N positions. Call the positions in X private and the remaining positions public; there are n private positions and N positions in all. Probability of choosing a public position in a random choice is around $(1-n/N)$ (this is an approximation because the number of public and private positions change as a result of a choice). Probability of choosing all public positions in t choices is $(1-n/N)^t$. So, the probability of choosing at least one private position is $1-(1-n/N)^t$. With $N, n = 10^6,\ 32$ the probabilities are .031, .062, 0.617, 0.959 for $t = 1000$, 2000, 30000 and 100000, respectively. Therefore, a fake author has to invert around 10% of the bits of the original file to assure removing an imprint; for larger files a larger fraction of the bits have to be inverted if n remains unchanged. It is unlikely that a file that differs in so many bits from the original can simulate the original file in any reasonable sense.

Remarks on the contents of certificates The certificate issued by CA should contain additional statistical information about a file such as its length and checksums for certain blocks. In the absence of such information, a pirate may truncate a genuine file drastically to remove the author's imprint, embed its own imprint by tweaking the truncated file and submit the truncated file for certification. The certificate issued to the truncated file is valid for the original file as well. Hence, the pirate can market the original file with the certificate issued for the truncated file.

3.3.3 Drawback of certification

The major difficulty in using certification is that privately shared files are no longer permitted; all such files have to be certified by a CA. Therefore, this scheme is best used for music and video files, that are, typically, publicly marketed.

4 Licensing from the Retailer

In this section, we present a few proposals for the retailer to issue licenses on behalf of the authors. There are many commercial reasons why customers may prefer to purchase files from a retailer rather than directly from the author. A retailer may handle files produced by several authors. For the retailer to issue licenses it must have the headers for all the files it sells, and it must sign a pair $(f.id, m)$ with the signature of f's author. Given honest retailers this scheme is immune to piracy in the sense described in theorem 1. However, a dishonest retailer may create unauthorized copies of files and issue licenses which are never reported to the authors.

A possible defense against dishonest retailers is as follows. An author associates a unique serial number with each copy of each file it sells. The serial number appears in plain text. The customer buys the file (on a CD, for instance) from the retailer and then contacts the author electronically for a license. The request for the license includes the serial number of the file as well as the machine id. The author issues the license, and it keeps track of the serial numbers sold. Then the author can demand the appropriate payment from the retailers based on the sales data and the serial numbers of the files issued to each retailer. Also, the author will not issue duplicate licenses for the same serial number. We reject this solution, however, because customers often buy from retailers when they have no convenient electronic contact with the author.

Electronic contact between a retailer and an author is more easily established. Therefore, a retailer may obtain the license from the author, on behalf of a customer, as described in the previous paragraph; the customer has to supply the machine id to the retailer. This procedure is analogous to "credit-card sale authorization" where a point-of-sale terminal (*pos*) obtains an authorization code for a credit card sale. The customer presents a machine-id card and the file to be purchased at a pos and the pos issues a license after contacting the author.

Another possibility is for the retailer to use a secure device, called a *licenser*. Presented with a file title, its author and a machine id, the licenser issues the appropriate license *and records the sale on permanent storage*. The device has to be secure so that the sales record can not be modified by the retailer. The storage requirement for a licenser are minimal: for each file sold by a retailer, it has to store the header, the signature function of the author and the number of copies of the file sold so far. The processing requirements are also minimal: for each sale a license has to be computed. Information on sales data may be read

out and reset by each author electronically (by using a secure authentication protocol) from time to time; therefore, only periodic contact between an author and a retailer is required. Since it is impossible for the retailer to access or manipulate any information in the licenser all sales of a file will be reported to its author.

In some countries such a device is already used for "fiscal journalling", i.e., the sales are recorded on a secure device for tax purposes.

Creating a license does not place extraordinary burden on a retailer. He may link the licenser to the point of sale terminal; the customer may provide his machine id in a smart card that can be scanned, and the file title can be obtained from its UPC code; the license can be printed in the sales receipt. Note that the license has no value to any other party; therefore, it may be given in plain text.

5 Extensions

Software leasing The scheme proposed in this paper permits leasing of software for a specific time period. Then, the license carries the terms of the lease, and these terms can be checked by the kernel as part of the license verification. The licenser device at the retailer has to record the lease term as part of the sale.

This scheme could be compromised by tampering with the clock in the machine. We assume that the clock is part of the kernel, and, hence, it is tamperproof.

Authorizing the customer vs. authorizing the machine We have, so far, assumed that a file is authorized for execution on a specific machine. This presents difficulties when the machine is sold to a different party; the buyer acquires and the seller relinquishes all licensed software on the machine. Instead, a file may be sold to a customer provided that the customer has a unique, unforgeable id encoded in a smart card that plugs into any machine. Authenticity check is then made against a customer id instead of a machine id. It is possible to authorize either a machine or a customer using this scheme. This allows a customer to run his software on any machine, but if the cards are truly unforgeable then a file licensed to a customer could be operating on at most one machine at any time.

Software refund It is difficult to carry out the following transaction: the customer returns a file and demands a refund. This is because he can keep a copy of his license and a copy of the original file.

Software reselling It is difficult for one customer to sell a file, that he had purchased, to another customer (and remove it from his machine). Reselling is a more general case of software refund (refund involves selling to the retailer), which is difficult, as argued above.

Bulk Licensing A file is sometimes licensed to run on any machine in a group, particularly, when software is licensed to an organization. Then, a single license may include the identities of all machines in the group, instead of licensing each machine individually. In particular, a special machine id, "ALL", could be used in a license for a privately shared file to enable it to be installed on any machine.

Acknowledgment

This paper owes a great deal to discussions with Rajeev Joshi, Don Fussell and Doug Burger. I am grateful to the members of WG 2.3 who gave me useful comments after presentation of this material. In particular, Ernie Cohen and J.R. Rao have given me expert advice on several aspects of this problem. The similarity of fiscal journalling to the use of licenser as well as the necessity of a secure clock for software leasing are due to Rao.

References

[1] Marc A. Kaplan. IBM Cryptolpes™, SuperDistribution and digital rights management. Available at http://www.research.ibm.com/people/k/kaplan, Dec 1996. IBM Corporation.

[2] R.L. Rivest, A.Shamir, and L. Adelman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[3] Yossi Shiloach. A fast equivalence-checking algorithm for circular lists. *Information Processing Letters*, 8(5):236–238, 1979.

[4] Olin Sibert, David Bernstein, and David Van Wie. Securing the content, not the wire, for information commerce. Available at http://www.star-lab.com/secure-the-content.html. Intertrust Technologies Corporation.

[5] A. J. M. van Gasteren and W. H. J. Feijen. Shiloach's algorithm, taken as an exercise in presenting programs. *Nieuw Archief Voor Wiskunde*, xxx(3):277–282, 1982.