

Surreptitious Software: Models from Biology and History

Christian Collberg^{1,*}, Jasvir Nagra^{2,**}, and Fei-Yue Wang³

¹ Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA
`christian@collberg.com`

² Dipartimento di Informatica e Telecomunicazioni, University of Trento, Via Sommarive 14, 38050 Povo (Trento), Italy
`jas@nagras.com`

³ Key Lab for Complex Systems and Intelligence Science, Institute of Automation, Chinese Academy of Sciences, ZhongGuanCun East Road 95, Beijing, Haidian, People's Republic of China
`feiyue@gmail.com`

Abstract. Over the last decade a bewildering array of techniques have been proposed to protect software from *piracy*, *malicious reverse engineering*, and *tampering*. While we can broadly classify these techniques as *obfuscation*, *watermarking/fingerprinting*, *birthmarking*, and *tamper-proofing* there is a need for a more constructive taxonomy. In this paper we present a model of *Surreptitious Software* techniques inspired by defense mechanisms found in other areas: we will look at the way humans have historically protected themselves from each other and from the elements, how plants and animals have evolved to protect themselves from predators, and how secure software systems have been architected to protect against malicious attacks. In this model we identify a set of primitives which underlie many protection schemes. We propose that these primitives can be used to characterize existing techniques and can be combined to construct novel schemes which address a specific set of protective requirements.

Keywords: Software protection, defense mechanisms, taxonomy.

1 Introduction

Your computer program can contain many different kinds of secrets that you may feel need to be protected. For example, you may want to prevent a competitor from learning about a particularly elegant algorithm. You therefore *obfuscate* our program, i.e. make it so convoluted and complex that reverse engineering it becomes a daunting task. Or, you may want to bind the copy sold to each person who buys it to prevent them from illegally reselling it. You therefore *fingerprint* the program, i.e. embed a unique identifier into each copy you sell,

* Supported in part by the Institute of Automation, Chinese Academy of Sciences.

** Supported by the European Commission, contract N° 021186-2, RE-TRUST project.

allowing you to trace a pirated copy back to the original purchaser. Or, you may want to prevent a user from running a program after he has manipulated it, for example by removing a license check. You therefore *tamperproof* the program, i.e. make it unexecutable/self-destructing/self-repairing if it detects that its code has changed. Or, you may want to detect if part of your program has been incorporated into your competitor's program. You therefore check for *birthmarks*, unique characteristics of your code, within your competitor's code.

These techniques have collectively been referred to as *intellectual property protection of software*, or *software protection*, or *whitebox cryptography*. However, we will henceforth refer to the area as *Surreptitious Software*.

Over the last decade many algorithms have been proposed to protect secrets in programs. Seeing as the area has been (and is still) in a great deal of flux, a core set of ideas and techniques on which these algorithms are built has yet to be identified. It is the purpose of this paper to serve as a starting point for constructing such a classification scheme. Our goal is to identify a set of primitives which can be used to build algorithms protecting secrets in programs, and to use these primitives to model and classify software protection schemes that have been proposed in the literature. It is our hope that this model will provide a uniform language for researchers and practitioners, making it easier to discuss existing protection schemes and to invent new ones.

In software engineering, researchers have developed the concept of "design patterns" [1] to capture the rules-of-thumb that regularly occur during the development of large pieces of software. Garfinkel [2] also describes user-interface design patterns for security applications. The models of attacks and defenses we will describe in this paper are similar. Our motivation for identifying and classifying software protection schemes is to eliminate the need to develop new schemes from first principles. Instead we seek to model attacks and defenses that occur repeatedly so experiences and solutions can be reused. We hope that as a result, the insights gained from defending against any one instance of an attack can be generalized to the entire class of defenses.

We will seek inspiration for this model from defense mechanisms found in nature, from the way humans have protected themselves from each other and from the elements, and from protection schemes found in software systems. We will see how, since the dawn of time, plants, animals, and human societies have used surreptition to protect themselves against attackers, and then see how (or if) these ideas can be applied to the intellectual property protection of software.

The model we present here is still in its infancy. In particular, to complement our model of the techniques used by the *defender* we're still working to model the techniques used by the *adversary*. Our ultimate goal is a model which will allow us to classify a proposed new software protection scheme as

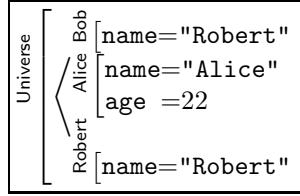
1. a simple variant of another, previously published scheme, or,
2. a novel combination of two known schemes, which we can predict will have certain properties, or
3. a novel scheme not fitting the model, forcing us to reconsider the model itself.

Ideally, we would like the model to *predict* new ways of protecting software, i.e. we should be able to deduce from the model if there are combinations of techniques that will produce a protection scheme with a particular set of required properties.

The remainder of the paper is organized as follows: In Section 2 we describe our notation and in Section 3 the intents of an attacker. In the main section, Section 4, we present the protection primitives available to the defender. In Section 5 we summarize our model and point out directions for future research.

2 Notation

Our model consists of a universe of objects. The attacks we are concerned with and defenses against these attacks are modeled as a series of transformations on these objects. We represent all objects as *frames*. Frames are knowledge representation devices well-known from the AI community. A frame consists of a unique identifier and one or more *slots*, which are *name=value*-pairs. We use a graphical notation to illustrate frames and frame transformations:



Each frame is introduced by a left square bracket (\square), and the object's unique identifier is written vertically to the left of the bracket.¹

The slots of a frame describe properties of an object. A value can be a scalar such as a string or an integer, a list of objects, another frame, or a reference to another frame. A special list slot is the *contains* slot, which, when present, indicates that a particular object contains other objects. The *contains* slot is indicated by a left angle bracket (\langle). In the example above the universe itself is such a frame, where the *contains* slot has three objects, **Alice**, **Bob**, and **Robert**.

In our model, protection strategies are functions which map frames to frames. We start out with a universe of unprotected objects and proceed by a sequence of protective transformations that make the objects progressively safer from attack. Layering protection schemes is modeled by function composition of the frame transformations. *Attacks* are also modeled by frame-to-frame transformations: they try to subvert protection strategies by *undoing* layers of protection.

As an example, consider the Leatherback turtle (*Dermochelys coriacea*) which buries its eggs in the sand to protect them from predators and the elements. In Figure 3 (a), the initial, unprotected scenario is a frame where the universe contains a *turtle*, *sand*, and *eggs*. Then, we apply the **cover** protection primitive,

¹ In AI, frames can be of various types called "abstractions". For example, "Bob" can have "human" and "attacker" abstractions. However, for our purposes of this paper we do not require this additional classification.

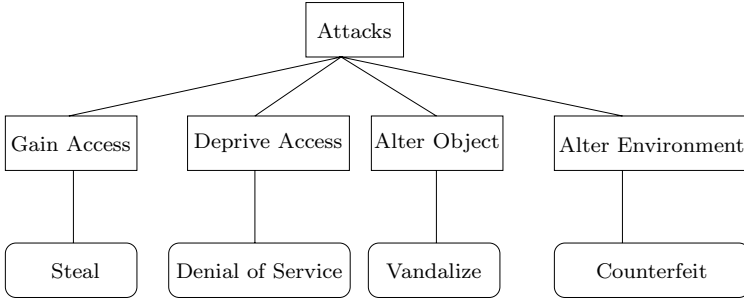


Fig. 1. Types of attacks

the sand frame is given a *contains* slot, and this slot now holds the turtle-eggs frame. This indicates that a predator can no longer see the turtle-eggs, them now being covered by sand. In other words, an object can only sense what’s in its own environment—it cannot look inside other objects.

Each slot can also have associated with it a set of *demons*. These are actions which fire under certain circumstances. For example, to protect its young the Cane toad (*Bufo marinus*) lays eggs that are poisonous. We would represent this form of protection with a demon attached to a toad egg saying “**when eaten then cause harm to attacker**”.

The model uses seven *basic operations* to construct frames and frame transformations. The **new** and **delete** operators create and destroy frames from the universe respectively. As in the real universe objects are never actually created or destroyed, but manufactured from existing material. We model this by giving **new** and **delete** access to a special *source* object which provides an infinite supply of raw-materials and a *sink* object into which excess material can be disposed. The **getprop**, **setprop** and **delprop** operators get, set and delete properties of a given frame. The **move** operator relocates an object represented by a frame from a particular position in one object to a different position within a possibly different object. Our final operator, **apply** is *function application*.

These seven basic operations allow us to build the universe of unprotected objects that we are interested in as well as the protective primitives that we describe. However, we do not give their explicit formulations in this paper.

3 Attacks

In order to describe protection schemes we require a corresponding model of the attacker. Protection is an act of defense against a perceived attack. The type of protection a scheme provides depends heavily on which types of attack are of interest. For example, in the case of the Leatherback turtle hiding its eggs, the threat being defended against is a predator stealing and eating the eggs, depriving the parent turtle of its offspring. In other cases, (like the Common Cuckoo, (*Cuculus canorus*)), an attacker may be interested in adding their own

eggs to the brood which appear similar to the nest owner's eggs. The intent of this attacker is to pass on their genes with minimal labor, by mimicking the object of value.

Given an object of value O , we can taxonomize the attacks by considering the different ways an attacker may derive value from O . The intent of an attacker may be to: gain unauthorized access to O , deprive legitimate users access to O , alter O , or alter the environment in which O occurs.

In Figure 1, a variety of attacks are shown classified into four different types. In a stealing attack, an attacker simply takes the object of value, while in a denial-of-service attack, an attacker deprives legitimate access to an object overwhelming it. Vandalizing a piece of property is an example of devaluing an object by altering it. Counterfeiting alters the environment of an object by filling it with imitations which in turn can reduce the value of the original.

The primitives that we will build in the next section are designed to prevent each of these types of attacks, irrespective of the context in which they occur. For example, in the next section we will introduce a **cover** operator. In the biological context, a Leatherback turtle can defend unauthorized access to its eggs by hiding them in sand using the **cover** operator. Similarly, in the historical context, pirates would hide their treasure by burying their treasure on a remote island using the same **cover** operator. In software, we may embed a picture inside a document to hide it from prying eyes. The document serves as a **cover** for the picture. In all three instances, the pattern of defense remains the same — hide the object of value using another object — even though the context may have changed.

4 Defenses

In Figure 2 we illustrate the eleven primitive operations of the defense model. Each primitive is a transformation from an unprotected universe of objects to a protected one. In the remainder of this section we will present each primitive in turn and illustrate with scenarios from biology, human history, computer security, and surreptitious software.

4.1 The Cover Primitive

The most fundamental way of defending against all four types of attacks is to hide an object by **covering** it with another object. If the attacker does not have access to an object, it becomes difficult for him to attack it. This may seem like a trivial observation, but **covering** is an operation that occurs frequently in the natural, human, and computer domains. In Figure 3 (a), we illustrate our earlier biological example of the Leatherback turtle. In Figure 3 (b), we show an historical example of **covering** occurring in 499 BC, when, to instigate a revolt against the Persians, Histiaeus tattooed a message on the shaved head of a slave, waited for the hair to grow back, and sent him to Axristagoras who shaved the slave's head again to read the message.

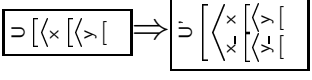
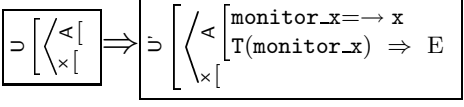
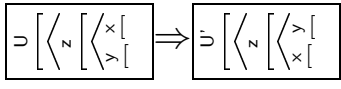
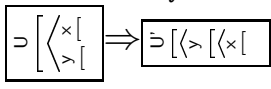
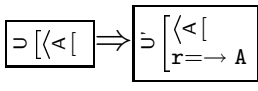
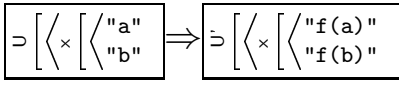
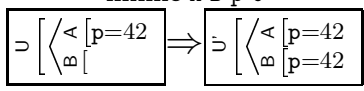
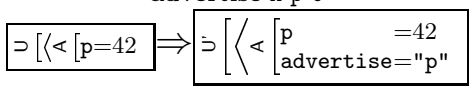
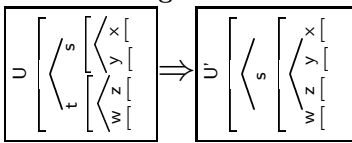
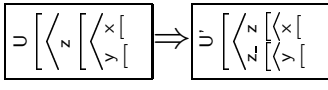
<p>compose f g $(f \circ g)(x) = f(g(x))$</p> <p>Compose primitives f and g.</p>	<p>dynamic x f U $x \Rightarrow f x \Rightarrow f(fx) \Rightarrow f(f(fx)) \dots$</p> <p>Repeatedly apply primitive f to object x in U.</p>
<p>copy x "-" U</p>  <p>Make of deep copy of x, adding "-" to keep names unique.</p>	<p>tamperProof A T x E U</p>  <p>When T happens to x, A takes action E.</p>
<p>reorder z f U</p>  <p>Reorder the elements of frame z according to function f.</p>	<p>cover x y U</p>  <p>Move x from its environment into/under y.</p>
<p>indirect A r U</p>  <p>r is an indirect reference to A.</p>	<p>map x f U</p>  <p>Apply function f to the components of x.</p>
<p>mimic A B p U</p>  <p>Copy property p from A to B.</p>	<p>advertise A p U</p>  <p>Make A's property p public.</p>
<p>merge s t U</p>  <p>Merge t into s, removing t from U.</p>	<p>split z f U</p>  <p>Split z into two parts, z and z_-. Function f returns True for elements which go in z.</p>

Fig. 2. Primitives of the protection model

In the software world, we can cover a file or some data by putting it inside another file. For example, when mail systems initially began scanning attachments for viruses, virus writers responded by zipping (and occasionally encrypting) the virus before emailing them. The zip file served as **cover** for the virus itself (Figure 3 (c)). In order to counter this cover, mail servers today have to “look under” every possible cover by unpacking and scanning every archive (including archives containing archives) to find potential viruses.

A more obvious instance of **covering** occurs in systems that use hardware to protect software. For example, the military uses hardened boxes to contain

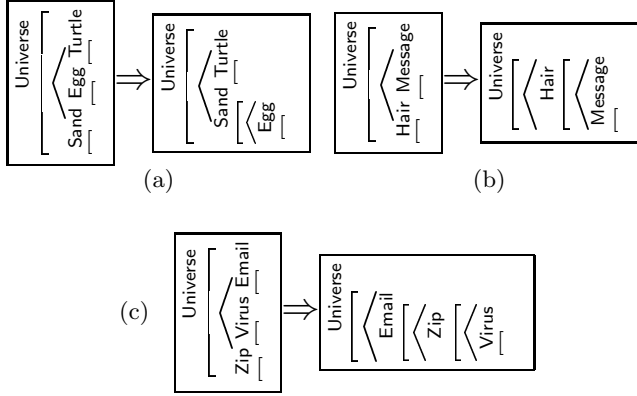


Fig. 3. Applying the **cover** primitive in the biological, historical, and software domains

computers on which sensitive software is run. These boxes are also **tamperproofed** so that they will explode (known by the military as “rapid disassembly”) if someone unauthorized tries to open them. **Tamperproofing** is in fact one of the primitives in our model, and we will discuss this further in Section 4.9.

4.2 The Copy Primitives

The **decoy** primitive makes the environment larger to force an attacker to consider more items while searching for a secret or to draw attention away from the real secret. The **clone** primitive adds a copy of the secret itself to the universe in order to force an attacker to destroy both copies. The difference between **clone** and **decoy** is merely one of intent, and in Figure 2 we therefore merge these primitives into one, **copy**.

In the animal kingdom, **decoying** and **cloning** are common protection strategies. An animal low on the food-chain will often use **cloning** to protect itself, or, rather it’s own DNA. For example, the Pufferfish (*Arothron Meleagris*) spawns 200,000 offspring in the hope that at least *some* will survive and carry on its parents’ legacy.

The California newt (*Taricha torosa*) combines **cloning**, **tamperproofing**, and **covering** to protect its progeny. The newt lays 7-30 eggs, each covered by a gel-like membrane containing tarichatoxin for which no known antidote exists.

Scientists are still unsure exactly why zebras have stripes, but one favorite theory suggests that they are used as part of a **decoying** strategy: when a zebra is running in a herd of similarly-striped animals it’s difficult for a lion to pick out one individual to attack. As we’ll see later, **mimicking** is another possible explanation.

Decoys are, of course, common in human warfare: “In World War II, the U.S. created an entire dummy army unit in southern Britain to convince the Germans that the Normandy invasion would be directed toward the Pas-de-Calais” and

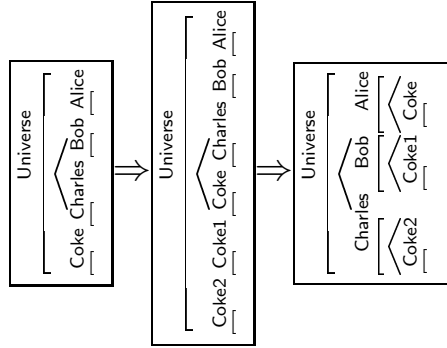


Fig. 4. Applying the **cloning** primitive to protect the Coca-Cola recipe. We’ve **cloned** twice and **covered** three times, resulting in a situation where the three executives are each holding a copy of the recipe in their pockets.

during Gulf War I, the Iraqis “employed mock tanks, airplanes, bunkers and artillery” made up of “plywood, aluminum and fiber glass” [3].

A well-known legend states that the recipe for Coca-Cola is protected by **cloning** it over three people, none of whom may ever travel on the same plane (Figure 4).

In the recent TV mini-series *Traffic*, terrorists attempt to smuggle smallpox virus into the United States. The authorities are conned into believing that the smallpox will enter the country inside a shipping container with a particular identification number, but are forced into a nationwide hunt once they realize that the terrorists have shipped several containers with *the same number*! This is a classic use of **decoying**, but later in this section you will see how the terrorists combined this ruse with the **advertise** primitive to further confuse the DEA.

Decoys can be used by software watermarking algorithms to protect the mark. In the very simplest case you just add a variable with a conspicuous name, like this:

```
int THE_WATERMARK_IS_HERE = 42;
```

This draws attention away from the actual watermark, if only for a short while. You could also add a number of fake watermarks, some really obvious, some less so, to force the attacker to spend valuable time examining them all, and having to decide which one is real and which ones are fakes. The **cloning** primitive can also be used to embed multiple copies of the mark (possibly by using different embedding algorithms), thus forcing the attacker to locate and destroy all of them.

In code obfuscation, **cloning** is also a common operation. For example, a function f could be cloned into f' , the clone obfuscated into f'' , and different call sites could be modified to either call f or f'' [4]. This gives the appearance that calls to f and f'' actually call different functions, both of which the attacker needs to analyze in order to gain an understanding of the program. A basic block B in a function can similarly be cloned into B' , obfuscated into B'' , and an

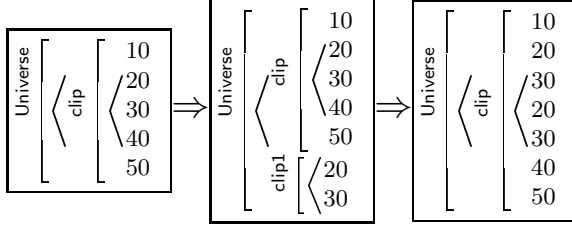


Fig. 5. Watermarking an image using echo hiding, by applying the **split** and **merge** primitives

opaque predicate can be inserted to alternate between executing the two blocks: `if $P?$ then B else B''` [5].

4.3 The Split and Merge Primitives

Split and **merge** are two common protection transformations, particularly in software situations where these are simple to implement. **Split** breaks up an object into smaller parts each of which are easier to hide or protect. **Merge** blends two unrelated objects together to make it appear as if they belong together. To create mass confusion, splitting and merging are often used in conjunction: take two unrelated objects A and B and split them into parts A_1 , A_2 , B_1 , and B_2 , then merge A_1 with B_1 and A_2 with B_2 .

When attacked, some animal species will split themselves and let the attacker have one part for dinner. This is known as *autotomy* [6]. This technique is particularly useful when the species is also able to *regenerate* the lost part. In our classification this is a form of **tamperproofing** which we will discuss further in Section 4.9 below.

Squirrels use so-called *scatter hoarding* to **split** up the food they've gathered and cache it at different locations in their territory. The larger Gray Squirrel (*Sciurus carolinensis*) is known to steal the caches of the European Red Squirrel (*Sciurus vulgaris*), but the **split** makes it less likely that all the food will be lost.

Human organizations frequently split themselves up into groups to prevent an adversary from destroying the entire organization in one attack. This is true, for example, of terrorist networks which split themselves up into smaller, autonomous cells. Each cell is less conspicuous than the whole network, and an attack (from an outside force or an inside informant) will affect only the cell itself, not the network as a whole.

Some software watermarking algorithms **split** the watermark into several smaller pieces such that each piece can be embedded less conspicuously [7,8]. **Merging** is also a natural operation, since the watermark object has to be attached to some language structure already in the program.

Echo hiding is an audio watermarking technique in which short echoes (short enough to be undetectable to the human ear) are used to embed marks. A really short echo encodes a 0 and a longer one a 1. In our model, embedding a single-bit watermark m is accomplished by a) settling on an embedding location

p , b) **copying** D_0 bits for a 0 and D_1 bits for a 1 from p , and c) **merging** the copy back into the clip. In this embedding example, shown in Figure 5, $D_0 = 2, D_1 = 3, p = 2, m = 0$.

Obfuscation algorithms make extensive use of **splitting** and **merging** to break up, scatter, and merge pieces of a program, thereby creating confusion. For example, to confuse someone trying to reverse engineer a program containing two functions f and g , each can be split in two parts (yielding f_a, f_b, g_a, g_b) and then the parts can be merged together forming two new functions $f_a||g_b$ and $f_b||g_a$. Any calls to f or g will have to be modified, of course, to call the new functions, and to call them in such a way that only the relevant parts are executed [4].

Two common protection strategies are **shrink** and **expand**. Shrinking oneself to become as small as possible makes it easier to hide in the environment. Expanding, on the other hand, is the operation of making oneself as large as possible, larger objects being harder to destroy than smaller ones. **Shrink** and **expand** are just variants of **split** and **merge**, however, so we don't include them as primitive operations. An object shrinks itself by splitting into two parts, one essential and one superfluous, disposing of the fluff. Expansion, similarly, is merging yourself with some new building material grabbed from the surrounding environment.

The Pufferfish uses a combination of several defense strategies. One is to **expand** its size (by inflating itself with water or air from its surroundings) in order to appear threatening or a less easy target to an attacker. The blue whale has grown so big that it only has one natural predator, the Orca.

That *size* is an important feature of hiding or protecting a secret is evident from the expression "*as hard as finding a needle in a haystack*." Implicit in this expression is that the needle is small and the haystack big. Hiding the *Seattle Space Needle* in a normal size haystack would be hard, as would hiding a sewing needle in a stack consisting of three pieces of hay. So, a common strategy for hiding something is to **shrink** it, such as a spy compressing a page of text to a microdot. This is often combined with **decoying**, increasing the size of the environment, for example using a really big haystack in which to hide the needle or hiding the microdot in the Bible instead of in the 264 words of the Gettysburg address. Expansion is, of course, also a common human strategy. A bigger vault is harder to break into than a smaller one, a bigger bunker requires a bigger bomb to destroy it, and in a crash between an SUV and a VW Beetle one driver is more likely to walk away unscathed than the other.

The principle behind LSB (Least Significant Bit) image watermark embedding is that the low order bits of an image are (more or less) imperceptible to humans and can be replaced with the watermarking bits. In our model, this is a **shrink** operation (shrinking the image gets rid of the imperceptible fluff) followed by **merging** in the watermarking bits.

It is important not to keep passwords or cryptographic keys in cleartext in computer memory. An attacker who gets access to your machine can search through memory for tell-tale signs of the secret: a 128-bit crypto key, for example,

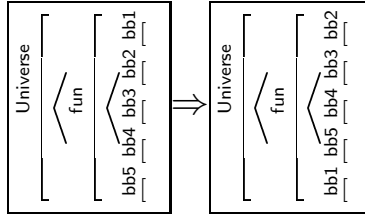


Fig. 6. Watermarking a program by applying the **reorder** primitive

has much higher entropy than ordinary data and will be easy to spot [9]. There are two simple ways of fixing this problem: either increase the entropy of the data surrounding the key (for example by encrypting it) or decrease the entropy of the key itself. The latter can be accomplished for example by **splitting** the key into several smaller pieces which are then spread out over the entire program. Thus, this is an application of the **split** and **reorder** primitives. We will examine **reordering** next..

4.4 The Reorder Primitive

The **reorder** transformation can be used to place objects in a random order, thereby *sowing* confusion. But a reordering can also *contain* information. Think of 007 entering a room and finding that the gorgeous blonde Russian operative who was just abducted left him a message: the martini glass has been moved to the *left* of the Baretta, letting James know that he needs to be on the lookout for Blofeld.

In various cons such as the *shell game* or *three-card-monty*, the secret (the *pea* or the *queen*, respectively) is hidden from the victim by sequences of deft **reorderings** by the con-man. This combines **reorder** with the **dynamic** primitive which we will see in Section 4.10.

Many watermarking algorithms, in media as well as software, make use of **reordering** to embed the mark. The idea is that a watermark number can be represented as a permutation of objects. For example, in the Davidson-Myhrvold software watermarking algorithm the watermark is embedded by permuting the basic blocks of a function’s Control Flow Graph. In the example in Figure 6, the watermark number 5 is embedded by ordering the permutations of the numbers [1 . . . 5]:

$$[1, 2, 3, 4, 5], [2, 1, 3, 4, 5], [2, 3, 1, 4, 5], [2, 3, 4, 1, 5], \underline{[2, 3, 4, 5, 1]}, \dots$$

picking the 5th permutation to reorder the basic blocks.

Many code obfuscation algorithms also make use of the **reorder** primitive. Even when there is an arbitrary choice on how to order declarations or statements in a program, programmers will chose the “most natural” order over a random one. Randomly reordering functions within a program will destroy the information inherent in the grouping of functions into modules, and the order of functions within modules.

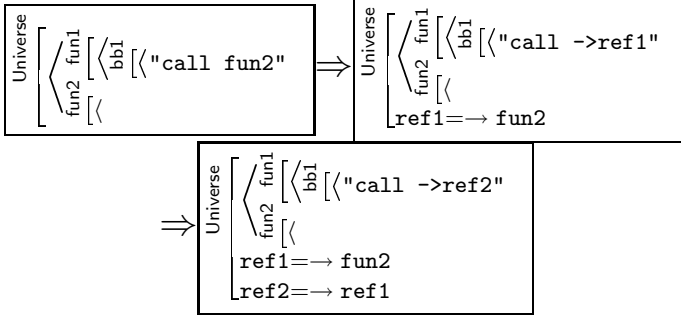


Fig. 7. Obfuscating a program by applying the **indirect** primitive

Of course, **reorder** can also be used as an attack: to destroy a watermark based on ordering all the attacker has to do is apply the algorithm again, destroying the previous ordering at the same time as he's inserting his own mark.

4.5 The Indirect Primitive

In baseball, the coach uses hand signs to indicate to the batter whether to hit away, bunt, etc. To prevent the opposing team from *stealing* the signs the coach will use **decoying** as well as adding a level of **indirection**. The actual sign is **merged** with a sequence of bogus decoy signs and a special *indicator* sign. The indicator gives the location of the actual sign, typically the one following directly after the indicator.

Indirection is also a common adventure movie plot device. In *National Treasure*, Nicolas Cage is lead by a sequence of increasingly far-fetched clues (a frozen ship in the arctic, a meerschaum pipe, the Declaration of Independence, a \$100 bill, etc.), each one pointing to the next one, eventually leading to the final location of the hidden treasure (a Freemason temple).

Like our other primitives, **indirection** is often used in combination with other protection strategies. Our hero will find that the next clue is inside a box hidden under a rock (**covering**), that there are many boxes under many rocks (**decoying**), that the box will explode unless he unlocks it just the right way (**tamperproofing**), and he that needs to find *both* pieces of the clue in order to locate the treasure (**splitting**).

One of our particularly clever friends came up with the following scheme for protecting her foreign currency while backpacking around Europe. First, she sewed the different currencies (this was before the European Union adopted the Euro) into different parts of her dress, a form of **covering**. Next, in order to remember where the Deutchmark, French Francs, etc., were hidden, she wrote down their locations on a piece of paper (**indirection**). Of course, if someone were to find this note she'd be in trouble, so she wrote it in French (**mapping**) using the Cyrillic alphabet (another **mapping**), banking on few common thieves being fluent in both Russian and French. She never lost any money.

Since exact pointer analysis is a hard problem [10], **indirection** has also become a popular technique in protecting software [5,11]. The idea is to replace a language construct (such as a variable or a function) by a reference to it. This adds a confusing level of indirection. Just like Nicholas Cage running around chasing one clue after another, an attacker analyzing a protected program would have to chase pointer-to-pointer-to-pointer until finally reaching the real object. Consider the example in Figure 7. Here we start out with two functions **fun1** and **fun2**, where **fun1** contains one instruction, a call to **fun2**. We apply the **indirect** primitive to add an indirect reference **ref1**, and then the **map** primitive to replace all references to **fun2** by an indirect reference through **ref1**. We then repeat the process to force an attacker to unravel two levels of indirections.

4.6 The Map Primitive

The **map** primitive typically protects an object by adding confusion, translating every constituent component into something different. If the mapping function has an inverse, this obviously needs to be kept secret.

Translation is a form of **mapping** where we map every word m in a dictionary to another word r , usually in a different language, keeping the mapping secret. The Navajo Code talkers are famous for their rôle in World War II, conveying messages in a language unfamiliar to the Japanese. Humans use this trick to confuse outsiders, or often to build cohesion among members of a group. Consider, for example, the 133t language used by youths in online communities. It sets them apart from others who don't know the language, but it also protects their communication from outsiders (such as their parents).

Obfuscated language mappings occur in many professional fields as well, including computer science, medicine, and law. Steven Stark [12] writes in the Harvard Law Review that “*one need not be a Marxist to understand that jargon helps professionals to convince the world of their occupational importance, which leads to payment for service.*” In other words, obfuscating your professional language protects you from competition from other humans of equal intelligence, but who have not been initiated into your field.

The most obvious way to protect a secret is to not tell anyone about it! This is sometimes known as *security-through-obscurity*. An example is the layout of many medieval cities in the Middle East. At first it may seem that there is no good reason for their confusing alleyways until you realize that the lack of city planning can actually be a feature in the defense of the city. Without a map only those who grew up in the city would know how to get around — an obvious problem for attackers. But this strategy only works in an environment with poor communications; as soon as there's a *Lonely Planet* guide to your city your attempts to use secrecy to protect yourself will have failed.

Figure 8 shows the simplest form of code obfuscation, *name obfuscation* [13], a form of translation that replaces meaningful identifiers with meaningless ones.

A common way of watermarking English text is to keep a dictionary of synonyms, replacing a word with an equivalent one to embed a 0 or a 1 [14]. See Figure 9 where we've used a dictionary to embed a 1 at position 2 in the text.

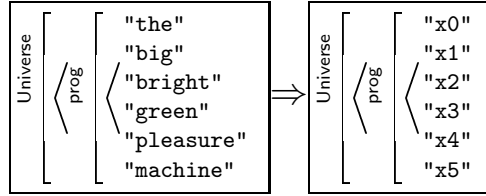


Fig. 8. Applying the **map** primitive to rename identifiers

Cryptographers use the terms *confusion* and *diffusion* to describe properties of a secure cipher [15]. Confusion is often implemented by *substitution* (replacing one symbol of the plaintext with another symbol) which, in our terminology is a **mapping**. Diffusion is implemented by *transposition* (rearranging the symbols), what we call **reordering**. A *product cipher* creates a secure cipher from **compositions** of simple substitutions and transpositions.

4.7 The Mimic Primitive

Mimicry is, of course, the greatest form of flattery. We can use it as many different forms of protection. *Camouflage*, for example, is trying to look like an object in your environment. Mimicry can also be a deterrent — you can try to look like someone or something you’re not, to scare off attackers. In our model, the **mimic** primitive simply copies a property from one object to another.

The chameleon is probably the animal most well-known for use of camouflage, **mimicking** the colors of items in its background in order to avoid detection. Many other animals avoid predators in a similar way. As we noted earlier, scientists are not sure exactly why the Zebra has stripes. One theory is that lions (being color blind) cannot pick out a zebra from the background when it stands still in high grass, i.e. the zebra **mimics** it’s background. Another theory contends that the stripes confuse Tse-tse flies (*Glossina palpalis*).

Peter Wayner’s *mimic functions* [16] create new texts which steganographically encode a secret message. In order not to arouse suspicion, the texts are made such that they **mimic** texts found in our daily lives, such as transcripts of a baseball announcer, shopping lists, or even computer programs. If the mimicry is good, i.e. if the statistical properties of a generated text are close enough to those of real texts, then we will be able to send secret messages across the Internet without anyone noticing.

The same technique has been used by spies. For example, during World War II, Japanese spy *Velvalee Dickinson* wrote letters about her doll collection to addresses in neutral Argentina. When she was writing about dolls she actually was talking about ship movements. She was eventually caught and jailed.

In 2004, during a vehicle search, Hungarian police found what appeared to be a fake Viagra tablet. Apart from being pink, it was rhombus shaped, had Pfizer imprinted on one side and VGR50 on the other — clearly a bad Viagra counterfeit. However, further analysis revealed that the tablet was just **mimicking**

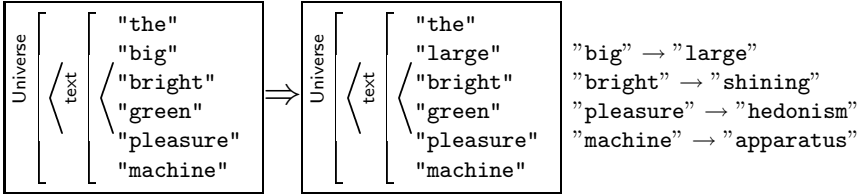


Fig. 9. Applying the **map** primitive to embed a watermark in English text

Viagra and instead contained amphetamine. See Figure 10. Here we’ve applied the **mimic** primitive three times, once for each attribute. Why the drug smugglers did not also **mimic** the highly recognizable blue color of Viagra tablets is hard to know, but it’s possible that they thought that appearing to be a bad counterfeit would make it less likely for the tablet be tested for illegal substances.

A central concept in surreptitious software is *stealth*. Any code that we introduce as a result of the protection process must fit in with the code that surrounds it, or it will leave tell-tale signs for the attacker to search for. In other words, the new code must **mimic** aspects of the original code written by humans. There can be many such aspects, of course: the size of the code, the instructions used, the nesting of control structures, and so on, and a stealthy protection algorithm must make sure that the introduced code **mimics** every aspect. By embedding a watermark in a modified **cloned** copy of an existing function, Monden’s [17] software watermarking algorithm **mimics** the style of code already in the program, thereby increasing stealth. As seen in Figure 11 this is actually accomplished using a combination of the **clone** and the **map** primitives.

4.8 The Advertise Primitive

By default, our model assumes that objects keep all information about themselves secret. In other words, Alice can see what other objects are around her, but she only knows their identities, she can’t look *inside* them. This is how we normally protect ourselves: We call the primitive which breaks this secrecy **advertise**. In our model it is represented by a special property **advertise** which lists the names of the properties visible to the outside.

One way **advertising** assists the defenses of an object is by identifying itself to its defenders. Flamingoes (*Phoenicopterus ruber ruber*) bring up their young in large crèches, however, parents feed and take care of only their own chicks. The chicks **advertise** their appearance and vocalization. These characteristic allow parents to recognize and defend their own progeny. In software, similar birthmarks [18] — unique characteristics of a program — allow an author to detect piracy by recognizing characteristics of their own programs in other software.

Another common use of this primitive is for the defender to advertise the strength of his defenses in the hopes that an adversary will stay away. There is nothing, of course, that says the defender must advertise *correct* information! In fact, a common use for this primitive is to lead an attacker astray by feeding them falsehoods.

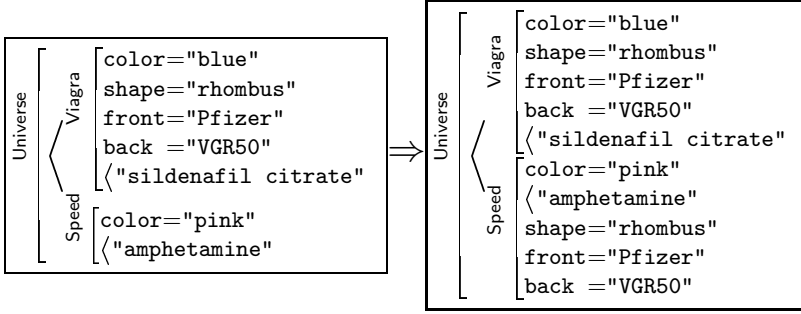


Fig. 10. Using the **mimic** primitive to protect an illegal drug from detection

Toxic species often use bright colors to advertise their harmfulness. This is known as *aposematic coloration* [6]. The easily recognizable red-yellow-black striped coloration pattern of the highly poisonous Coral snake is an example of this. The King snake protects itself by **mimicking** the Coral snake, while actually being non-venomous. Falsely **advertising** “Look, I’m venomous!” will confuse predators to stay away. The transformations are shown in Figure 12.

In many ways, **advertise** can be seen as the opposite of *security through obscurity* — rather than keeping our defenses secret we openly display them to our adversary, taunting them to “go ahead, take your best shot!” In an ideal scenario they will take one look at us, walk away disheartened, and launch an attack against a less well defended target. In a less ideal scenario, knowing details of our defenses will allow them to work out a chink in our armor - like a 2 meter exhaust vent leading straight into the core of our Death Star allowing anyone with a spare proton torpedo to reduce us to so much space debris.

We already mentioned the recent TV mini-series *Traffic*, where terrorists smuggle smallpox virus into the United States, sending the authorities on a wild goose chase across the country hunting for the right container among several **decoy** containers with the same number. In the end, it turns out that the terrorists had used **advertise** (“the virus is in one of these containers **wink, wink**”), to trick the DEA: *all* the containers were actually decoys and the actual carriers of the virus were the illegal immigrants on board the cargo ship.

False **advertising** also works in the software protection domain. A program can advertise that it is watermarked, when in fact it isn’t, or **advertise** that it is **tamperproofed** using a particular algorithm, when in fact it is using another one.

4.9 The Tamperproof Primitive

Tamperproofing has two parts, detecting that an attack has occurred and reacting to this. The reaction can be a combination of

1. self-destructing (in whole or in part),
2. destroying objects in the environment (including the attacker), or
3. regenerating the tampered parts.

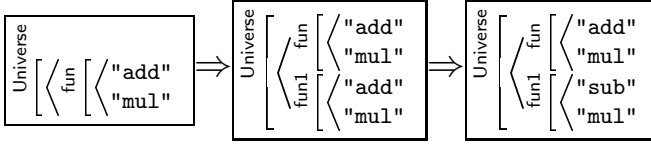


Fig. 11. Stealthy software watermarking using **mimicry**. We first **clone** an existing function and then **map** its instructions to new ones to embed the mark.

Some animals can **regenerate** destroyed parts of their bodies (usually limbs and tails but in some cases even parts of internal organs) after an attack. Starfish can regrow their entire organism given just one arm and the central disk.

A Pufferfish uses a combination of several defense strategies: he can **expand** his size to appear threatening to an attacker, and an attacker who never-the-less starts munching on him will have to deal with the neurotoxins of his internal organs, a form of **tamperproofing**. It may not save an individual Pufferfish's life, but it might save his brothers and sisters since the attacker is either dead or has developed a distaste for Pufferfish.

Turtles and many other animals use exoskeletons (a form of **covering**) to protect themselves. Some combine this with **tamperproofing**, using poisons to make themselves unpalatable once the protective layer has been removed by the predator. The Hawksbill turtle (*Eretmochelys imbricata*) has both a shell and poisonous flesh.

Humans using tamperproofing to protect themselves is a common movie plot: the terrorist straps explosives to his chest and holds a dead-man's-trigger in his hand so that if the hero takes him out everyone will die in the ensuing blast. Another common plot device is leaving an envelope containing compromising information about your nemesis with your lawyer, with instructions that "in the case of my untimely demise, mail this to the Washington Post."

The terrorist in the above scenario is *self-tamper-detecting* as well as *self-tamper-responding*. This seems to be the most common case: you monitor some part of yourself (the health of your internal organs, for example), and if tampering is evident you execute the tamper-response (releasing the trigger a split second before expiring). The second scenario above shows that both detection and response can be external, however: the lawyer examines the obituaries every day and, if your name shows up, executes the tamper-response.

Unlike lower order animals like newts and salamanders, humans have very little regenerative power — we can only regrow some skin and a part of our liver.

Tamperproofing is, of course, an extremely important part of surreptitious software. A common form of tamper-detection is to compare a hash computed over the program to the expected value [19]. Some tamperproofing algorithms make use of *regeneration* [20], to fix parts of a program after an attack. The idea is to replace a destroyed part with a fresh copy, as shown in Figure 13.

Other types of response are also common: a program can retaliate by (subtly or spectacularly) self-destructing [21] or destroying files in its environment. For

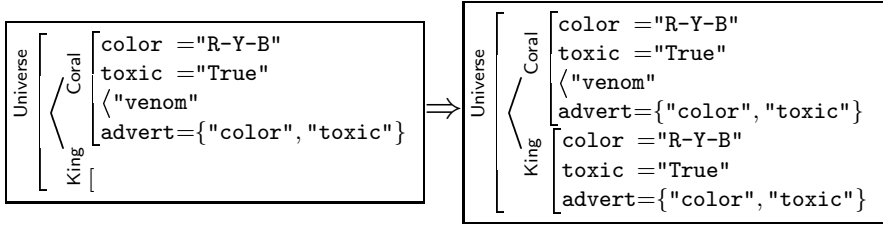


Fig. 12. The King snake applies the **advertise** primitive to fool predators that it contains venom, even though it does not. (**advertise** is here abbreviated **advert**.)

example, the *DisplayEater* screen motion catcher program will delete your home directory if you try to use a pirated serial number [22].

4.10 The Dynamic Primitive

Our final primitive, **dynamic**, is used to model protection techniques which use dynamic behavior to throw off an attacker. Here, *continuous change* itself is used to confuse an attacker. The change can take a variety of forms: fast movement, unpredictable movement, and continuous evolution of defenses.

Moving faster than your adversary is in many ways the ultimate protection technique in the animal world. If the cheetah can't catch you, he can't eat you! There is a trade-off between agility and speed on the one hand and physical layers of protection on the other. A turtle can afford to be slow because he's carrying around a thick protective exoskeleton. The Pronghorn antelope (*Antilocapra americana*) has made a different trade-off: it can run 98 km/h to get away from a predator such as the mountain lion which is only able to do 64 km/h. On the other hand, the antelope is soft and vulnerable on the outside and should he get caught, it's game over.

Anyone who has tried to kill a particularly annoying fly knows that while speed is important, so is agility and unpredictability. Consider, for example, the fruit fly (*Drosophila Melanogaster*) which flies in a sequence of straight line segments separated by 90 degree turns, each of which it can execute in less than 50 milliseconds. Since the movements appear completely random, a predator will have to be very lucky and persistent to get the best of him.

Evolution itself is, of course, an exercise in continuous change. As prey develop better defenses, predators develop better attacks. HIV is one example of a particularly successful organism: because of its fast replication cycle and high mutation rate, a patient will experience many virus variants during a single day. This makes it difficult for the immune system to keep up and for scientists to develop effective vaccines.

Mobility is key in modern warfare. During the Gulf War, Iraqi forces successfully moved Scud missiles around on *transporter-erector-launcher* trucks to avoid detection by coalition forces: "even in the face of intense efforts to find and destroy them, the mobile launchers proved remarkably elusive and survivable" [23]. Just like the Pronghorn antelope and the turtle have made different

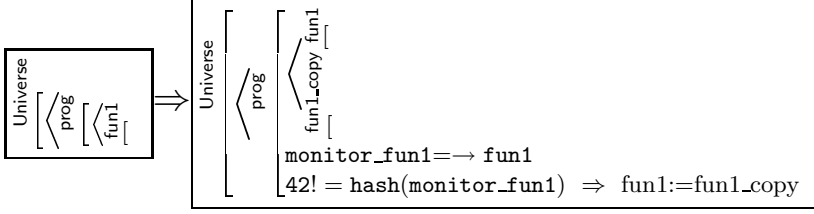


Fig. 13. Using **tamperproofing** to regenerate a part of a program that has been tampered with

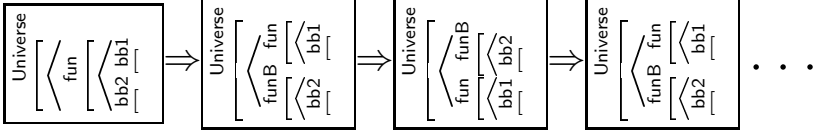


Fig. 14. Aucsmith's tamperproofing algorithm [24] uses the **dynamic** primitive

trade-offs on the scale from *slow-but-impenetrable* to *fast-but-vulnerable*, so have military forces.

Just like in the natural virus world, in the computer virus world change is used to foil detection. A computer virus that modifies itself for every new generation will stress the capabilities of the virus detector: it has to find some signature of the virus that does not change between mutations.

Many software protection algorithms are dynamic, in one form or another. Aucsmith's tamperproofing algorithm [24], for example, first breaks up a program in chunks which are then encrypted. One chunk is decrypted, executed, re-encrypted, swapped with another chunk, and the process repeats, as shown in Figure 14. (Here we're not modeling the encryption. This could be done with the **map** primitive.) This process results in an unpredictable address trace that an attacker will find difficult to follow, making the code hard to tamper with.

The **dynamic** primitive is a higher order function taking an object x and a function f as arguments, iteratively generating $\langle fx, f(fx), f(f(fx)), \dots \rangle$.

5 Summary

It's an interesting exercise to come up with scenarios — biological, historical, or computational — that can't be expressed using the primitives we've suggested in this paper. Also interesting is to show that a particular primitive can be simulated by a combination of others, and thus eliminated from the model. We invite the community to join us in improving the model in this way.

It should be kept in mind that any Turing complete model (such as a simple Lisp-based model) would do, but the goal here is not to come up with a minimalist set of primitives. Rather, we want to find primitives which elegantly express how the natural world has evolved protection strategies, how we as humans think

about protecting ourselves and, most importantly, of course, how we can protect computer programs.

We are currently establishing an *open experimental environment for surreptitious software*, where various malicious attacks, protection techniques, and test programs from any interested parties can be created, launched, and can interact with each other according to certain protocols, leading to a “living laboratory” for evaluating software security through computational experiments [25].

References

1. Cunningham, W., Beck, K.: Using pattern languages for object-oriented programs. In: OOPSLA’87 (1987)
2. Garfinkel, S.: Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (2005)
3. unknown: Decoys: Tanks but no tanks. Time Magazine (Monday, Feb. 4) (1991), www.time.com/time/magazine/article/0,9171,972244,00.html
4. Collberg, C., Thomborson, C., Low, D.: Breaking abstractions and unstructuring data structures. In: IEEE International Conference on Computer Languages 1998, ICCL’98, Chicago, IL (1998)
5. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 1998, POPL’98, San Diego, CA (1998)
6. Cronin, G.: Defense Mechanisms. Salem Press, pp. 314–319 (2001)
7. Collberg, C., Huntwork, A., Carter, E., Townsend, G.: Graph theoretic software watermarks: Implementation, analysis, and attacks. In: Workshop on Information Hiding, pp. 192–207 (2004)
8. Collberg, C., Carter, E., Debray, S., Kececioğlu, J., Huntwork, A., Linn, C., Stepp, M.: Dynamic path-based software watermarking. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 04), ACM Press, New York (2004)
9. Shamir, A., van Someren, N.: Playing Hide and Seek with stored keys. In: Franklin, M.K. (ed.) FC 1999. LNCS, vol. 1648, pp. 118–124. Springer, Heidelberg (1999)
10. Ramalingam, G.: The undecidability of aliasing. ACM TOPLAS 16(5), 1467–1471 (1994)
11. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia (2000)
12. Stark, S.: Why lawyers can’t write. Harvard Law Review 97(1389) (1984)
13. Tyma, P.: Method for renaming identifiers of a computer program. US patent 6,102,966 (2000)
14. Bender, W., Gruhl, D., Morimoto, N., Lu, A.: Techniques for data hiding. IBM Syst. J. 35(3-4), 313–336 (1996)
15. Shannon, C.E.: Communication theory of secrecy systems. Bell System Technical Journal, 656–715 (1949)
16. Wayne, P.: Mimic functions. CRYPTOLOGIA 14(3) (1992)
17. Monden, A., Iida, H., Matsumoto, K., Inoue, K., Torii, K.: A practical method for watermarking Java programs. In: 24th Computer Software and Applications Conference (2000)

18. Myles, G., Collberg, C.: k-gram based software birthmarks. In: Proceedings of SAC (2005)
19. Horne, B., Matheson, L., Sheehan, C., Tarjan, R.E.: Dynamic self-checking techniques for improved tamper resistance. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, Springer, Heidelberg (2002)
20. Chang, H., Atallah, M.: Protecting software code by guards. In: Sander, T. (ed.) DRM 2001. LNCS, vol. 2320, Springer, Heidelberg (2002)
21. Gang Tan, Y.C., Jakubowski, M.H.: Delayed and controlled failures in tamper-resistant systems. In: Information Hiding 2006 (2006)
22. Farrell, N.: Mac Display Eater kills home files. *The Inquirer* (February 27, 2007)
23. Keaney, T.A., Cohen, E.A.: Gulf War Air Power Survey Summary Report (1993)
24. Aucsmith, D.: Tamper resistant software: An implementation. In: Anderson, R. (ed.) Information Hiding. LNCS, vol. 1174, pp. 317–333. Springer, Heidelberg (1996)
25. Wang, F.Y.: Computational experiments for behavior analysis and decision evaluation of complex systems. *Journal of Systems Simulations* 16(5), 893–897 (2004)