Preventing Piracy While Preserving Privacy

A flexible antipiracy solution

MICHAEL O. RABIN AND DENNIS E. SHASHA

In the battle between pirates and content providers, the pirates are winning. Movies appear on bootlegged DVDs and on peer-to-peer networks even before they appear in theatres. Expensive software can be obtained at rock-bottom prices without royalties flowing to the authors. Pricey technical counter-measures are easily defeated. In 2002, a multimillion dollar CD-based antipiracy scheme developed by Sony, was defeated by writing on the outer rim of protected CDs with a magic marker. License servers are routinely cracked. Total losses, while hard to calculate exactly, may amount to tens of billions of dollars per year.

Content vendor reactions vary from hand-wringing to threats of lawsuits to hope for yet a better protected medium. Platform vendors such as Intel, Microsoft, Apple, and Panasonic are more ambivalent. If one platform prevents piracy, will consumers choose another? This proposition has not been tested but platform vendors have been cautious so far. Some content vendors even view piracy as a kind of loss leader. A few years ago, a scientist from a leading vendor, for example, announced to an expert panel (in substance): "Piracy doesn't worry us. The best thing that can happen to us is that someone buys our software, next that someone steals it, and the worst that someone buys our competitors' software." More recently, however, a scientist from the same company said to one of us: "We can no longer afford to sell just one copy in country X and see the rest stolen."

Frustrated with platform vendor inactivity, content vendors have chosen to use law enforcement and the courts to stop piracy—261 lawsuits were filed on just one day in 2003, for example. This has met with some success, but only in some countries and in a few cases. Even then, there is something distasteful about prosecuting librarians and 12-year-old children.

There must be a better way. Look beyond computer software and beyond movies to driving behavior. When faced with speed bumps, you slow down. You don't need police to tell you to. Your butt or your passengers' discomfort will ensure you don't speed. The underlying philosophy behind our solution is to implement a software speed bump to combat piracy. Our solution requires no police and preserves the privacy of everyone, even pirates.

Michael is a professor at Harvard University and a recipient of the Turing Award. Dennis is a professor at New York University and is the puzzle columnist for DDJ. *They can be contacted at rabinmichael@aol.com and shasha@cs.nyu.edu, respectively.*

As a matter of terminology, we use the term "software" (or simply "content") to indicate any digital content, such as computer programs, computer games, audio and video, and so on.

Starting Points

We start with two assumptions, the first moral and the second technical.

> "The underlying philosophy behind our solution is to implement a software speed bump to combat piracy"

The moral assumption is that stealing is wrong, even if it's easy. A screenplay writer friend once said she doesn't condone stealing except of computer software. It weighs nothing. The computer copies it. Some big corporation suffers. What could be wrong? How would she feel if someone stole her screenplay? Oh, that's different. But it isn't, big corporation or not. She wouldn't feel it's okay to steal a car from an automobile factory.

On the other hand, the punishment should fit the crime. Ideally, software pirates should get no benefits from pirating but there should be no jail sentences or onerous fines. The more successful we are at preventing piracy by technical means, the less the need for law enforcement and high penalties. (As a matter of legal principle, if the odds of getting caught are 1 in 1000, then the penalty should be 1000 times the profit to render piracy unattractive. We avoid high penalties by reducing the profit from piracy to virtually zero.) Think speed bump again.

The technical assumption is that User Devices (computers or other software playing devices) have a secure clock and software called the "Supervising Program" that cannot be changed and that is given a periodic time-slice when it can run. "Secure" means that even the owner of the device cannot alter the progress of

(continued from page 16)
the clock, alter the Supervising Program, or intervene with its actions. This assumption lies within the technical state of the art:

- In a paper written in 1992, Lampson, Abadi, Burrows, and Wobber [1] suggested a way to load an operating-system kernel reliably using a bootstrapping method based on a single cryptographic key. Continuous checking of the integrity of the kernel or of our Supervising Program can be achieved by similar means. In these days of inexpensive hardware, there are other possibilities—IBM, HP, and Dell already ship computers that include a so-called "Trusted Platform Module," a coprocessor providing a feature called "remote attestation" [2]. The Trusted Platform Module can guarantee (and even promise to other devices) that a certain operating system and a certain BIOS are running. Similar techniques can be used for the Supervising Program. Only the Supervising Program needs to be secure, not the Software that is later going to be protected.
- Hardware vendors must provide a clock that advances continuously and uniformly (for example, one that keeps in step with Greenwich Mean Time so is unaffected by time zone or daylight-savings time). The Trusted Platform Module already provides a counter that is guaranteed to increase over time.
- The operating system must interact with the Supervising Program by ensuring that it runs periodically.

The Shield system does the rest, ensuring piracy prevention while preserving privacy. Before we discuss it, however, we briefly examine the main existing approaches to prevent piracy.

Current Approaches to Combat Piracy

Many companies offer piracy prevention or, more generally, digital-rights-management software. The main distinction between the two is that digital-rights-management software may also include linguistic constructs to describe usage possibilities, a prominent example being ContentGuard's XrML language (http://www.contentguard.com/xrml.asp). In this article, however, we concentrate on piracy prevention, because that is the fundamental technology upon which all else rests.

The best current approach is to encapsulate software inside hardware. Video cameras do this, but in the computer software world, such software comes on hardware attachments, such as so-called "dongles," like those from MicroWorks (http://www.mw-inc.com/) and SafeNet (http://www.safenet-inc.com/products/tokens/ikey1000.asp). This solution is feasible if the dongle can be rendered tamperproof and by running impractical-to-reconstruct parts of the software program on the dongle. The dongle approach is vulnerable to a reverse engineering attack of that "impractical-to-reconstruct" software. Even when the dongle approach works technically, however, the hardware approach makes it difficult to use several unrelated but protected software items at once and is, in general, cumbersome.

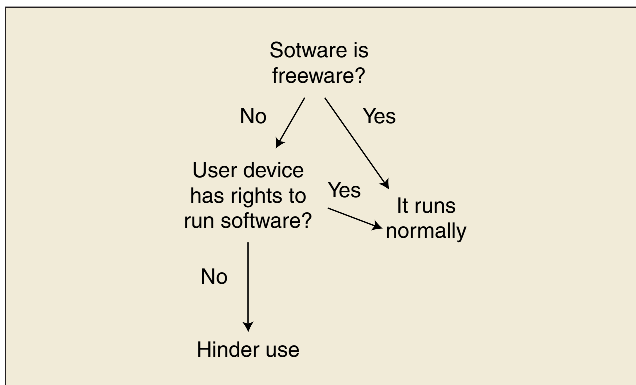

Figure 1: *Piracy prevention flowchart. No information leaves the User Device.*

A part-hardware approach is to ship software out on "copyproof" CDs. Again, extremely low-tech attacks (scribbling on CD rims) have defeated such solutions in the past. But even if the CD is truly copyproof, what happens if the content ends up on a web site from which it can be downloaded. This attack, dubbed "Break-Once, Run Everywhere" (BORE), can render an entire factory's work a waste of time and effort.

A software-imitates-hardware approach is to encrypt the content and ship the key to the client site, which can then execute the software only if it has the proper keys. This solution suffers from the BORE problem as well: If the content can ever be constructed in the clear through either an attack on the encryption, an attack on even one User Device where the software has been running, or an insider leak by an employee of the software author, it can be used everywhere.

License servers combat piracy by requiring licensed software to get permission to continue running from time to time. This scheme can be attacked if a would-be pirate can simulate the license server's responses, or change the software not to query the license server. If either happens, there is a BORE problem. In addition, this solution requires the software author to modify the software by introducing the (hopefully nonremovable) calls to the license server. Even if not, the notion of having to report usage to an outside license server inherently infringes on privacy.

There are approaches that don't try to prevent piracy but try to track and/or punish the pirates. The "watermarking" approach is to write some unique undetectable digital message on each instance of the software. If that digital message is found on many instances of software in the field, then the original purchaser of that watermarked copy is the source of those copies. The problems with this scheme range from the theoretical (it doesn't seem possible to create an undetectable watermark) to the practical (how does one track down copies and test them for watermarks). Further, there is the problem of legal punishment. Trials are expensive, time-consuming affairs. Finally, the technique depends fundamentally on violating privacy, because it requires identifying the "criminal."

A second form of punishment is to put "poisoned apples" in places where pirates are likely to look. The idea is to punish pirates by giving them something that looks good but isn't—conceivably a virus but more commonly a broken piece of content. Two years ago, a pirate downloading a Madonna song from a site might instead find a furious Madonna piping out expletives. Since then, poisoning peer-to-peer networks has become a thriving cottage industry.

For certain kinds of software, notably movies and music, the aforementioned solutions do not prevent a would-be pirate from digitally recording the content while watching or listening and then later redistributing the recording. Copying and redistributing content in this way is known as the "Analog Hole" attack.

All existing solutions (other than wrapping the software inside a hardware device) suffer from a BORE attack. Most of these solutions infringe on privacy, sometimes by design. A better solution should avoid BORE, avoid courts, and preserve privacy.

Towards a New Approach

Our approach to protection is simple: As Figure 1 illustrates, periodically during the execution of software on the User Device, our Supervising Program checks whether the software is freeware or not. If not, the Supervising Program identifies the software and checks whether this User Device has the rights to run this software. If so, the software continues to run; if not, the software is either stopped or markedly slowed down. No information leaves the device. The punishment is to hinder use.

To realize this approach, we have to specify how rights are transported to the User Device, how rights can be transferred

(continued from page 18)
between User Devices for purposes of fair use and upgrades, and how the Supervising Program can determine which software is running. At each step, we show how privacy is preserved.

The basic data flow of the Shield system is in Figure 2. Briefly, privacy-preserving purchases are shown on the left side of the User Device, content-identifying information enters from the Superfingerprint server depicted on the upper right, and privacy-preserving rights information is exchanged with the Guardian Center.

One important point: The indicated interactions with the Software Vendor, the Superfingerprint Server, and the Guardian Center are infrequent (on the order of once per week) and need little bandwidth. People who like to work mostly offline can continue to do so.

Privacy-Preserving Purchase

Our ability to preserve privacy while preventing piracy is based on the fact that rights, as embodied in "Tags," are stored on the User Device in data structures called "Tag Tables"; see Figure 3. The relationship between the Tag Table Identifier (TTID) and the Tag is an internal affair of the User Device. At purchase time, Tag-related information flows between the User Device and the Vendor/Author, but the Vendor/Author does not know for which TTID. At rights-management time, TTIDs flow between the User Device and the Guardian Center but the Guardian Center does not know for which Tags. So even if the Vendor, Author, and Guardian Center all collude, they cannot determine which sets of Tags belong to the same User Device, much less which particular User Device owns any particular Tag.

When the owner of a User Device wishes to purchase digital content (including digital content that has been preloaded on the User Device or installed from a CD), the Supervising Program on that device creates a structure identifying the software and its associated Tag Table Identifier:

S = (Name(C), TTID, Hash(C), UsagePolicy, NONCE)

Name(C) is the name of the content. *TTID* is the identifier of the Tag Table into which the Tag will eventually go. *Hash(C)* is the hash value of the content. *UsagePolicy* is some kind of policy such as perpetual use or three-month use. *NONCE* is a number that is randomly chosen from a large number space (for instance, from 128-bit numbers) and that is never used again. We use the *NONCE* to hide the value of *TTID* even should the Vendor collude with the Guardian Center.

A Purchase Order consists of:

(Hash(S), Name(C), Hash(C), UsagePolicy)

The hash function is a one-way hash function (see the accompanying text box entitled "Crypto Technologies") such as SHA-1 (or any of its improved versions), so no outsider can compute the *TTID* by inverting the function and no outsider can guess-and-check the *TTID* because of the *NONCE* component of *S*.

The Purchase Order may be sent to Vendor/Author over an anonymizing network, to make the source unknown to the sender [3]. The purchase may be in digital cash. Thus, the Vendor/Author can be prevented from knowing the identity of the purchaser, but can verify that the purchase amount corresponds to the correct price.

If so, the Author digitally signs the Purchase Order *Sign_Author(PurchaseOrder)* and sends it to the User Device via the Vendor. (By signing, the Author guarantees that it is paid for every purchase. If Vendor signatures were sufficient, then a rogue Vendor could start selling content on its own. As a practical matter, the Author may devolve signing privileges to select Vendors.)

The Supervising Program then verifies that the Author's signature is correct. This is possible because the User Device has previously downloaded from the Superfingerprint Server authenticated (digitally signed) data including a list of the Authors' public signature-verification keys. If the signature is verified to be that of the author of the content *C*, and *S* is consistent with the signed Purchase Order, the Supervising Program installs the triple (Author name, *S*, signed Purchase Order) into the Tag Table having identifier *TTID*. That triple is the Tag; see Figure 4.

If a user pays with anonymous digital cash (or even a one-use credit card) and sends orders over an anonymizing network (see, for example, http://tor.eff.org/), the Vendor/Author will not know who did the purchase. Further, the Vendor/Author will not know which TTID is associated with this purchase.

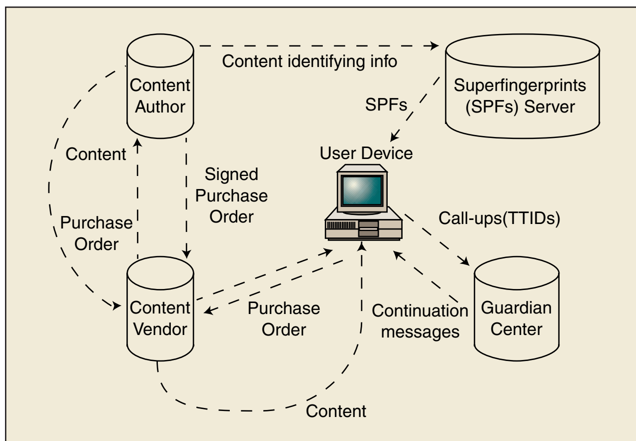

***Figure 2:** System architecture.*

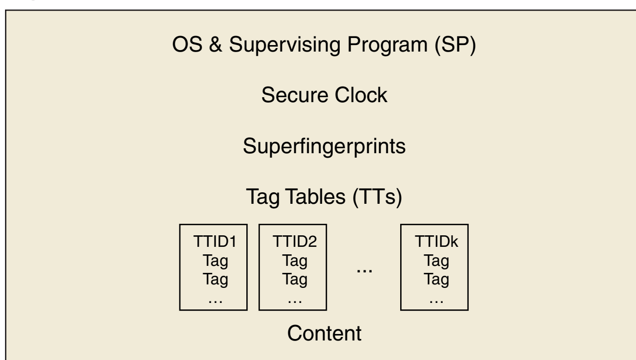

***Figure 3:** User Device.*

Superfingerprint Information

The Superfingerprint Server (upper right corner of Figure 1) periodically sends several kinds of information updates to the User Device. All User Devices receive the same information and must be reasonably up-to-date (for instance, this information must not be more than one week old, so the User Device must receive the Superfingerprints once a week).

- Content Identifying Information. This data associates with the name *Name(C)*, of each content *C* that is protected by the system, data enabling the Supervising Program to identify *C* when it runs. What running or executing means depends on the type of digital content. In the case of a computer program, running means the execution of the program and identification information can then be derived from sequences of machine instructions executed by the program at runtime and from functionalities of the program. Alternatively, the content could be music, in which case, the identification information could be derived from frequency components of the melody. The Content-Identifying Information for a content *C* typically fits

(continued from page 20)
in about 1/1000 of the number of bytes of *C* (significantly less for movies).

Each Author wishing to protect a content *C* runs a program (or asks a professional organization to run a program) that generates relevant Content-Identifying Information. That information is distributed to Superfingerprint Servers. These in turn send the additional Content-Identifying Information to User Devices during the next Superfingerprint broadcast. A Vendor/Author need not change the content *C* in any way to enable this protection. As a consequence, the antipiracy protection can be deployed after distribution of *C*.

- Content-Identifying Algorithms. The Supervising Program initially includes a suite of Content-Identifying Algorithms (which employ the Content-Identifying Information) to identify protected content. The algorithms are tailored to the type of content; for example, one class of algorithms for computer programs, another for music or video, and so on. But the algorithms apply to all examples of content in each class.

  One attack on the combination of Content-Identifying Information and Content-Identifying Algorithms consists of obfuscating the code or music or other content so it has the same effect to end users but looks different to our detection system. Experimentation has shown that detection algorithms can be made robust against a wide range of obfuscation attacks. (Compressing the content does not hinder our detection because detection occurs primarily at runtime.) The framework counters further obfuscation attacks by requiring the User Device to obtain periodic updates (weekly, for instance) of Content-Identifying Information and algorithms from the Superfingerprint Server. As obfuscations improve, so can our detection.
- Lists of pairs: Signature-verification key, Author name. This information lets the User Device verify whether a given Author's signature corresponds to an Author. In addition, there will be pairs relating the hashes of content to Author names. Together, these ensure that the signature of an Author as found in a Tag in fact constitutes sufficient Authority to allow the use of software. This combats the attack where author *A* creates content *X* but author *B* signs Purchase Orders for content *X* without having the right to do so.

All communication with the Superfingerprint Server is one-way—from Superfingerprint Server to User Device, again possibly through an anonymizing network. Consequently, no information leaves the User Device.

Transfers Without Promiscuity

Finally, there is the question of managing rights. Fair rights laws and tradition require the ability to make backups. Our technology

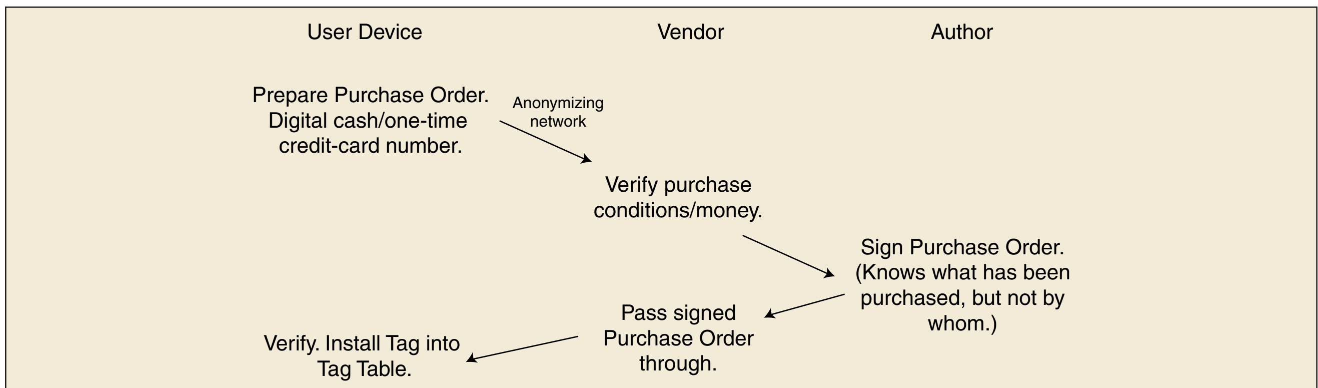

Figure 4: *Privacy-preserving purchase. Identity of user hidden by anonymizing network and digital cash. Tag Table Identifier is embedded into Purchase Order using a one-way function.*

Crypto Technologies

Whereas our approach never encrypts content, it makes substantial use of three cryptographic technologies—one-way functions to hide Tag Table Identifiers and User Device Descriptive Values, digital signatures to establish the identity of sites on the network, and Secure Sockets Layer (SSL) to ensure private communication of TTIDs.

Intuitively, a function f is one-way if, given x, it is easy to compute $f(x)$ whereas given y, it is hard to find an x such that $y = f(x)$. The hash function SHA-1 is one example (among many) of a one-way function.

The purpose of a digital signature is the same as of a written one—to establish the identity of the signer of a message. When you sign a contract, the holder of that contract can go to court and assert your agreement to the contract. Ideal written signatures are unforgeable but recognizable: only *X* can produce *X*'s signature but anyone can recognize that signature. So, only one person can sign, but anyone can verify (at any time or place). Digital signatures work the same way: An agent (say, the Guardian Center) in our protocol uses a private key to sign a document but that agent's signature-verification key is well known (say, is in the Supervising Program of every User Device). Therefore, if a message arrives purporting to be from that agent, then any User Device can test whether the message is in fact from that agent.

The Secure Sockets Layer (SSL) protocol is a client-server protocol offering asymmetric authentication and private communication. SSL assures the client (in our protocols, the User Device) that the server has a particular identify (in our Call-Up protocol, that the server really is the Guardian Center). SSL also enables the client and server to agree on a private key, which can be used in subsequent communication. The net effect is that the client knows the identity of the server (but not the other way around) and that the content of the exchange between client and server remains hidden from anyone else.

—M.R. and D.S.

allows any number of backups to be made of everything— the Tags, Tag Tables, and content.

Further, we want to allow transfers of rights, so Tag Tables may be moved from one User Device to another, provided the Tag Table is disabled on the first device. On the other hand, we don't want the same Tag Table to appear on millions of devices. We reconcile these two goals through communication between each User Device and the Guardian Center. The basic purpose of this communication is to determine whether a Tag Table having some Tag Table Identifier is on several devices.

Let us back up for a moment. TTIDs come about by randomly generating an identifier from a large (128-bit numbers) space perhaps based on time, typing characteristics, or a special random process. The chances of collisions in such a case are, for all practical purposes, negligible until the number of TTIDs is extremely large (for instance, a billion billion for 128-bit TTIDs). So when first created, every Tag Table has a globally unique TTID.

To ensure that only one User Device contains a particular TTID at a given time, each User Device performs a "Call-up" between some minimum and maximum time, say every five to seven days. As shown in Figure 5, a Call-up from device *U* consists of a message to the Guardian Center where the message contains a list of all enabled TTIDs of User Device *U*, a timestamp, and the hash of a "User Device Descriptive Value" of *U* appended to a *NONCE*. The User Device Descriptive Value contains some slowly changing property of the device that only a small number of devices have (for example, a processor ID, if available, or something about the number of files or structure of directories on the device). The use of the one-way hash function prevents any knowledge of this value from leaving the device.

The Call-up is sent using a well-known secure protocol such as SSL (see "Crypto Technologies"), so no third party can see which TTIDs are being sent. The Guardian Center checks each TTID *x* in the list of TTIDs to see whether an overly recent Call-up contained *x*. If so, the Guardian Center either records the fact for future reference or, if this has happened more than some threshold number of times, the Guardian Center invalidates that TTID.

After this analysis, the Guardian Center responds to the Call-up with a signed "Continuation Message" listing valid Tag Table Identifiers:

```
Sign_GuardianCenter(timestamp, Hash(User Device Descriptive
                     Value, NONCE), TTID1, TTID3,...)
```

The timestamp ensures that the device cannot simply replay an old Continuation Message. The hash together with the *NONCE* prevent the Guardian Center from learning the User Device Descriptive Value. The User Device Descriptive Value permits the Supervising Program on User Device *U* to ensure that the Continuation Message was meant for *U*. This prevents a single Continuation Message from being used by many shadow User Devices.

The User Device associates the most recent Continuation Message and its associated User Device Descriptive Value with each Tag Table. If the User Device Descriptive Value no longer matches the relevant properties of the User Device (perhaps due to a transfer of a Tag Table to this device), the Supervising Program on the User Device performs a new Call-up for just that Tag Table.

On the User Device, the Supervising Program disables Tag Tables whose TTIDs have not been included in the most recent Continuation Message. There is a grace period policy, however, allowing devices to use the software associated with Tag Tables even if out-of-date, provided this doesn't happen too often.

A user transfers content by disabling its associated Tag Table *x* on the source device and sending it to a destination device. After doing a Call-up for Tag Table *x*, the destination device can now use all the software items whose Tags involve the transferred TTID.

Failure to disable Tag Table *x* and its TTID on the source device will soon thereafter lead to overly frequent Call-ups for that TTID being sent to the Guardian Center. Call-ups must be done over a secure channel (such as SSL) to prevent malicious users from faking Call-ups with a given TTID *y* just to deny the real owner of the Tag Table having TTID *y* from using that Tag Table.

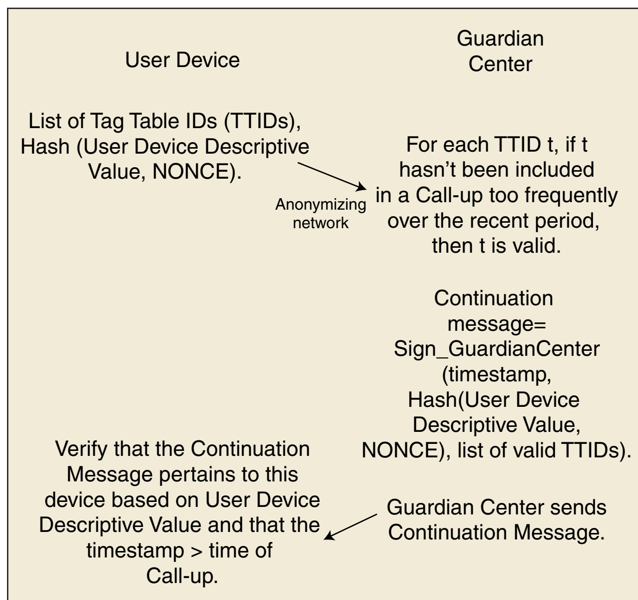

Figure 5: *Privacy-preserving Call-ups. User knows that it is talking to the Guardian Center but not vice versa (an option of SSL). TTIDs do not reveal the associated Tags. The one-way hash function associated with the* NONCE *prevents any revelation of the User Device Descriptive Value, so even processor identifiers can be used without fear of privacy breach.*

Note also that the Guardian Center need not be a single device. Guardian Center data may be replicated and any one of several Guardian Center nodes can handle a given Call-up request, or data may be partitioned based on TTID. (The Guardian Center data consists of information about TTIDs: time of last Call-up and a history of any overly early Call-ups.) In any case, the Guardian Center workload scales easily.

Putting It All Together

Here is a quick overview of the whole system. Every User Device includes a Supervising Program. When software *C* is being used (for example, executed) on the User Device, the Supervising Program attempts to identify *C* by use of Content Identifying Information and Algorithms present on the User Device. If unsuccessful, then *C* is deemed to be freeware and use proceeds. If identified as software named *N*, then the Supervising Program searches for a Tag for *N* in a Tag Table having a valid TTID. If found, then the Supervising Program verifies that the current usage is in accordance with the UsagePolicy for that instance of *C* included in the Tag for *C*. If everything checks out, then use of *C* is allowed, otherwise use is stopped or hindered.

The Supervising Program is run at regular periods, checking the running queue of the User Device. It can be designed to consume fewer than 2–3 percent of the computing resources. In our experiments, its impact on the performance of even compute-intensive workloads, such as computer games, is unnoticeable.

The Supervising Program performs the protected software installation task. The actual software purchase can be done outside of the User Device, for example, by an organization's purchasing department.

(continued from page 24)

The Supervising Program periodically downloads authenticated (that is, timestamped and digitally signed) updates of the Content-Identifying Information, Content-Identifying Algorithms and lists of (Author name, content-hash) pairs, and (Author name, signature-verification key) pairs from the Superfingerprint Server.

To revalidate its Tag Table identifiers, the Supervising Program periodically calls up a Guardian Center. The Call-ups are infrequent and require little bandwidth.

Transfers entail movements of Tag Tables from one User Device to another. Back-ups are unlimited. Every reasonable model of fair use is easy to implement. For example, it's possible to lend your software to your friend (two transfers), to allow short term use (Tags having short-term Usage Policies), and family packs (single purchase yields the privilege to obtain multiple Tags).

Frequently Asked Questions

When we talk about this framework, we hear several questions:

Q: How can we claim that we preserve privacy when we have Call-ups?

A: The Call-ups send information that neither identifies the user nor the software nor the Tags on the User Device, because TTIDs are sent rather than Tags. The protocol can be verified by third parties. Alternatively, you could avoid Call-ups by linking Tags to machines IDs, but then transfers would become more complicated and purchases as well as transfers might potentially infringe on privacy.

Q: Why don't we suffer from BORE? Superfingerprints detect use of software rather than mere possession. Can't one subvert your detection?

A: Maybe, but it is possible to do a very good job of detecting functional equivalents of software. Also, Superfingerprints can be improved with each download to counter new attacks.

Q: What happens when you catch someone stealing?

A: The Supervising Program on the device stops or slows down the use of that software. No information leaves the User Device. This is the functional equivalent of a speed bump: Behave, because you get car-sick if you don't.

Q: So, if this is so great, why isn't this adopted?

A: For this architecture to take hold, the hardware and operating-system Vendors must cooperate. The enabling technology for the protection system essentially exists, so it is a question of willingness. Platform Vendor incentives aren't so clear. If one platform Vendor provides piracy prevention and another doesn't, consumers may prefer the one that doesn't. If our solution is used, the only reason consumers will have to dislike the piracy-prevention system is that it prevents the ability to steal. It is possible that legislation will be necessary to ensure that no platform vendor benefits by making a platform that makes stealing easier. There is precedent for this: When catalytic converters to reduce automobile pollution emissions first came on the market, many consumers resisted their introduction because they made both acceleration and gas mileage suffer, besides raising the price of the car. Their introduction has greatly reduced air pollution, however, so it constituted a societal good. Legislation was necessary to avoid having consumers punish vendors who advanced that societal good. The same may happen here. Further, whereas our architecture imposes negligible penalties on performance, it permits many new usage models such as paying for use only when needed (pay for tax software only at tax time), the preloading of software, and digital distribution of software.

The saved costs from cheaper distribution and vastly reduced piracy run into tens of billions of dollars, enough to benefit all players—authors, consumers, and platform vendors. Again, there is precedent for the situation where taking on a burden ultimately enhances profit. When credit-card companies cap payments by consumers due to fraudulent uses of their cards, consumers feel more confident about using their credit cards. Similarly, when platform vendors support this framework, this will allow many new creative and inexpensive uses of and distribution of content, enhancing the value of platforms everywhere and ultimately reducing the price of software to all consumers. Indeed, we foresee an alliance between (enlightened) consumers, platform vendors, and authors supporting this framework, because it is in everyone's economic and artistic interest.

Conclusion

The Shield Approach is a flexible, privacy-preserving, antipiracy solution that does not suffer from "Break Once, Run Everywhere." It protects privacy in a strong sense: It can be configured so that no one knows what you buy, what you use, or even whether you cheat. Because the content is obtainable separately from the Tag, preloading the content is possible. Transfers and fair use are straightforward. Finally, the solution is technology friendly. We embrace peer-to-peer networks, video-on-demand, superdistribution, and free software. Content Vendors will feel free to distribute content over the Internet, reducing distribution costs and material waste. Lawsuits will be reduced. Isn't it time for technology to solve this problem?

Acknowledgments

Warm thanks to our principal coworkers in this effort: Yossi Beinart, Carl Bosley, Ramon Caceres, Aaron Ingram, Timir Karia, David Molnar, and Sean Rollinson.

References

[1] "Authentication in Distributed Systems: Theory and Practice." Butler Lampson, Martmn Abadi, Michael Burrows, and Edward Wobber. *ACM Transactions on Computer Systems* Volume 10, Number 4, (November 1992), pp. 265–310.

[2] For information about trusted coprocessors, see https://www.trustedcomputinggroup.org/home/.

[3] For information about anonymizing networks, see http://tor.eff.org/.

DDJ

Pragmatic Exceptions . . . .

Tip #2: Refactor for Exceptional Clarity

Tip #2 is the logical follow-on from Tip #1, "If In Doubt, Throw It Out," (*DDJ*, September 2005). In deciding if a local exception should be caught or pitched out, well-factored and highly focused code is ideal. Refactoring is fundamental to good programming practice in so many ways, an important one being that it helps you understand the finer points of the method contract. This removes doubt and leads to better decisions regarding exceptions.

The smaller your functions, the easier it is to tell whether what just happened was normal. Because each function can clearly specify exactly what should be expected, knowing whether to throw an exception becomes obvious. Large functions obscure or even obliviate the contract they're supposed to fulfill.

— Benjamin Booth
benjamin.booth@gmail.com