SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **00**:1–23
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

A Novel Watermarking Method for Software Protection in the Cloud

Zhiwei Yu[1,2], Chaokun Wang[2,3,4*,†], Clark Thomborson[5], Jianmin Wang[2,3,4], Shiguo Lian[6], and Athanasios Vasilakos[7]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing, China*
[2]*School of Software, Tsinghua University, Beijing, China*
[3]*Key Laboratory for Information System Security, Ministry of Education, Beijing, China*
[4]*Tsinghua National Laboratory for Information Science and Technology, Beijing, China*
[5]*Department of Computer Science,The University of Auckland, New Zealand*
[6]*France Telecom R&D (Orange Labs), Beijing, China*
[7]*University of Western Macedonia, Greece*

SUMMARY

With the rapid development of cloud computing, software applications are shifting onto cloud storage rather than remaining within local networks. Software distributions within the cloud are subject to security breaches, privacy abuses, and access control violations. In this paper, we identify an insider threat to access control which is not completely eliminated by the usual techniques of encryption, cryptographic hashes and access-control labels. We address this threat using software watermarking. We evaluate our access-control scheme within the context of a Collaboration Oriented Architecture, as defined by The Jericho Forum.

Received . . .

KEY WORDS: Cloud Computing, Software Protection, Watermarking

1. INTRODUCTION

With the rapid development of computer and Internet technology, illegal copying, tampering and other infringements on digital products are increasingly rampant. This reduces the profits of software manufacturers, distributors, and service providers, and it also affects governmental tax revenue. Legal sanctions against infringers are ineffective unless an infringement can be detected. Economic and social sanctions may also be deterrents, but as with legal remedies the infringement must be detected before an effective response can be invoked. Furthermore, when the copyright holder and suspected infringer are in different jurisdictions, economies, or social groupings, it is difficult for the copyright holder to enforce a sanction. For these reasons, researchers have been increasingly motivated to find technical means for detecting copyright infringements, and for responding to suspected infringements.

Figure 1 is an estimate of the prevalence of software license-infringement activity (a.k.a. "software piracy") in various parts of the world [1, 2]. According to this estimate, more than one-third of all installations of personal computer (PC) software is unlicensed. Current detection and response systems are apparently ineffective in some parts of the world. We note that these estimates are for software on home PCs as well as on commercial PCs. We imagine that commercial entities,

[†]E-mail: chaokun@tsinghua.edu.cn
[*]Correspondence to: Chaokun Wang, School of Software, Tsinghua University, Beijing, China

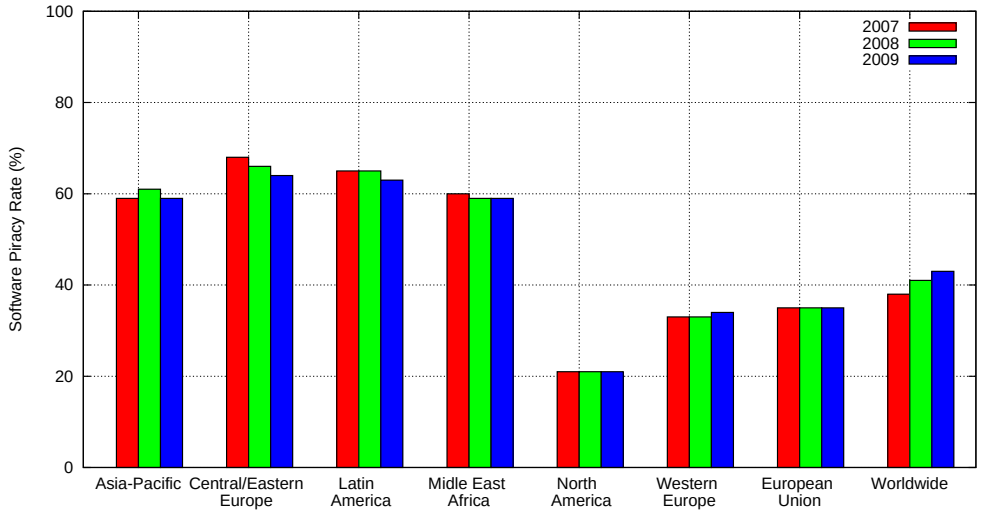

Figure 1. Software Piracy Rates : 2007-9

especially if they are reliant their reputation in overseas markets, would be very unlikely to use unlicensed software if infringements could be reliably detected and reported by some trusted entity.

In this article, we form a security model of the copyright-infringement problem by considering the security concerns of a pair of organisations who have agreed to collaborate through the cloud. One organisation provides software by uploading it to the cloud, and the other organisation uses this software after downloading it from the cloud. This is an IaaS (infrastructure as a service) cloud, because the organisations are using the cloud as a shared storage medium. In our concluding analysis, we also briefly consider an SaaS cloud, in which the software is run on cloud-based servers.

In the IaaS scenario, the outsider threat to software copyright could be effectively mitigated by encrypting the software under keys known to the two collaborating organisations. This mitigation of outsider threat will be complete if the keys are impossible to guess, if the keys are not revealed to outsiders, and if the decrypted software is developed and used on trustworthy platforms which are secure against eavesdropping attacks.

Insider threats are more difficult to counter. We can identify three classes of insider threat to software copyright, depending on the location of the insider. Insiders in the software-producing organisation may illicitly distribute the software for private gain or revenge. Insiders in the cloud infrastructure, and in the software-consuming organisation, are our other two classes.

When mitigating an insider threat, we must identify some trusted entity which will monitor or restrict the activities of the other (less trusted) insiders in the system. If our trust in the monitor is well-placed, the threat will be mitigated. If the monitor is untrustworthy (i.e. incompetent, inattentive, or venal), the insider threat will not be mitigated, and may even be exacerbated. This is a classical conundrum in security: “quis custodiet ipsos custodes” (who guards the guards)?

Advanced technology, such as modern cryptography, can lessen our dependence on the trustworthiness of human monitors, but we must rely on the trustworthiness of system designers, installers, maintainers, overseers, auditors, and any humans involved in the system’s response to a detected abuse. These people are carefully hired and managed in a security-conscious organisation; however when one organisation provides copyright software to another organisiation, the first organisation must find some way to monitor the security arrangements of the second organisation.

In this article, we explore the following technical means for the monitoring of one organisation by another. It is well-accepted that each instance of copyright software should have a visible label in, say, XACML. We argue that copyright software should, in addition, carry an invisible label which is embedded using software watermarking. The invisible label, if it is not consistent with the visible label, is an indicator of an insider attack.

DOI: 10.1002/spe

In our analysis, we concentrate on insider threats in the software-using organisation. We also consider insider threats from the cloud. Insider threats in the software-producing organisation are certainly important, but our watermarking system is ineffective against such threats. Software producers must, we believe, employ traditional safeguards such as physical security, inspections, hiring practices, security policies, and education.

We argue that the cloud is an appropriate location for a trusted entity which would mitigate insider threats in software-using organisations. The cloud has great powers of observation and control, with respect to the services and infrastructure it provides to its clients. The cloud must be trusted by our two organisations to deliver its store-and-forward service infrastructure with high availability and reliability, for otherwise the organisations would use more traditional methods for their software logistics. Here we suggest that the cloud, in order to be more fully trusted for software sharing, should offer an additional pair of services, that of embedding and detecting watermarks.

Others, notably the Cloud Security Alliance [3], have already argued that the cloud must be trusted, for otherwise it will not be used. Furthermore, the cloud must be trustworthy in all relevant ways, for otherwise it will be abandoned as soon as its security defects become manifest. Each organisation must make its own determinations of what aspects of the cloud should be trusted. The Cloud Security Alliance suggests asking the following six questions during a security assessment.

- How would we be harmed if the asset became widely public and widely distributed?
- How would we be harmed if an employee of our cloud provider accessed the asset?
- How would we be harmed if the process or function were manipulated by an outsider?
- How would we be harmed if the process or function failed to provide expected results?
- How would we be harmed if the information/data were unexpectedly changed?
- How would we be harmed if the asset were unavailable for a period of time?

Our focus in this paper is on the first of these questions: mitigating the threat of improper publication.

We are not aware of any IaaS cloud which is currently providing a detection service for improper publications of cloud-based assets. This may be because software watermarking is still an immature field. Accordingly, in this article, we provide some experimental evidence of the feasibility of software watermarking in the cloud. When preparing the current version of this article during the reviewing process, we discovered that other researchers have recently come to a similar conclusion: that the cloud is an attractive place to install an insider-threat detection mechanism based on watermarking [4, 5].

We claim three contributions in this paper. Firstly, we develop a model of insider threat for cloud-mediated collaborations involving the sharing of software. Secondly, we argue that this threat is inadequately mitigated using traditional techniques of (visible) security labelling and encryption, and we show how invisible (steganographic) labelling can improve the chances of preventing or detecting an illicit distribution. Thirdly, we implement a prototype of our scheme using the MapReduce programming framework and a "cloudlet". We use this prototype to evaluate our scheme in the context of the Collaboration Oriented Architecture [6] specified by The Jericho Forum.

2. RELATED WORK

2.1. *Existing Security Measures in the Cloud*

Infrastructural resources, platforms, and services are provided on-demand by dynamically provisioning hardware, software, and datasets in the cloud [7]. As a result, desktop computer users gain access to massive computational resources including databases, as well as a very wide range of networked services. However each new resource, each new platform, and each new service offers risks as well as benefits [3].

Security defects in the cloud have already caused significant problems for business and government. In 2009, social bookmarking site ma.Gnolia experienced a server crash that resulted

DOI: 10.1002/spe

in massive data loss, eventually causing the service to shut down forever. In 2008, a survey of 244 IT executives, CIOs and line-of-business colleagues revealed that security was the topic of greatest concern regarding their companies' use of IT Cloud Services, with 75% indicating that security was a significant or very significant issue that must be addressed before mainstream adoption [8]. The other issues polled were performance (63%), availability (63%), ease of integration (61%), insufficient ability to customize (56%), cost (50%), difficulty of bringing it back in-house (50%), regulatory requirements (49%), and not enough major suppliers (44%).

The first lines of defense in the cloud, as in any computer system, are strong encryption and digital signature in a carefully implemented and well-maintained system of access control. Non-technological methods such as education and legal remedies are also important [9, 10]. In the cloud, virtualised servers are generally used to limit the damage that can be done by rogue programs.

Hwang examines three major commercial cloud platforms for their security vulnerabilities: Google Cloud Platform, IBM Blue Cloud, and Amazon Elastic Cloud [11]. He recommends the use of reputation systems to keep track of security breaches at all levels of the cloud application stack. Our scheme for software watermarking would be consistent with this approach, for it would provide a way of detecting breaches.

Table I. Summary of Disclosure Threats and Corresponding Security Measures (after [12])

Threats in the Cloud		**Countermeasures**
Insiders	Inadvertent Disclosures	Alerts, Education, Training
	Intended Abuses	Encryption, Authentication, Authorization, Digital Signature, Auditing
Secondary Users	Unauthorized Distributions, Abuses	Encryption, Authentication, Authorization, Digital Signature, Firewalls and Network Service Management, Watermarking
Outsiders	Unauthorized Accesses	Encryption, Authentication, Authorization, Digital Signature, Firewalls and Network Service Management, Watermarking
Non-human Factors	Software and Hardware Failures	Software Management, System Vulnerability Analysis

Rindfleisch [12] identifies three types of disclosure threats and their principal countermeasures. In Table I, we augment his analysis by including a fourth threat of non-human factors, such as failures or bugs in software and hardware. We also mention some additional technologies, such as encryption and watermarking, that can be used as countermeasures. Here, outsider intrusions are defined as unauthorized accesses, either through the network or through physical attack. Outsider threat agents are angry customers, vindictive former employees, competitors, and thieves. Insiders may inadvertently release data, they may be curious, they may be bribed, or they may be greedy enough to betray the trust placed in them by their organisation. In our threat model, insiders are employees of the software-producing organisation, of the cloud service provider, and of the firm(s) who act as security auditors. A second tier of insider, represented by the software-consuming organisation in our threat model, are called "secondary users" in Rindfleisch's article. These semi-trusted threat agents can be controlled by a digital rights management system such the one described here, with oversight by external auditors and cloud service providers to lessen the threat of a disclosure abuse that is sanctioned by the software-consuming organisation.

To gain further insight into insider threats, we analysed three yearly reports [13] from the Identity Theft Resource Center of the United States. This non-profit organisation collates media reports from credible sources, and notification lists from state governments, regarding inadvertent or malicious breaches of personal identifying information. In accordance with U.S. Federal guidelines, they define a breach as "an event in which an individual name plus Social Security Number (SSN), driver's license number, medical record or a financial record/credit/debit card is *potentially* put at risk – either in electronic or paper format." In 2009, 458 breaches were reported, potentially exposing 222 million records. For 35% of the recorded breaches, no information is available (publicly) regarding how the breach occurred. The other breaches are caused by Hacking (20%),

DOI: 10.1002/spe

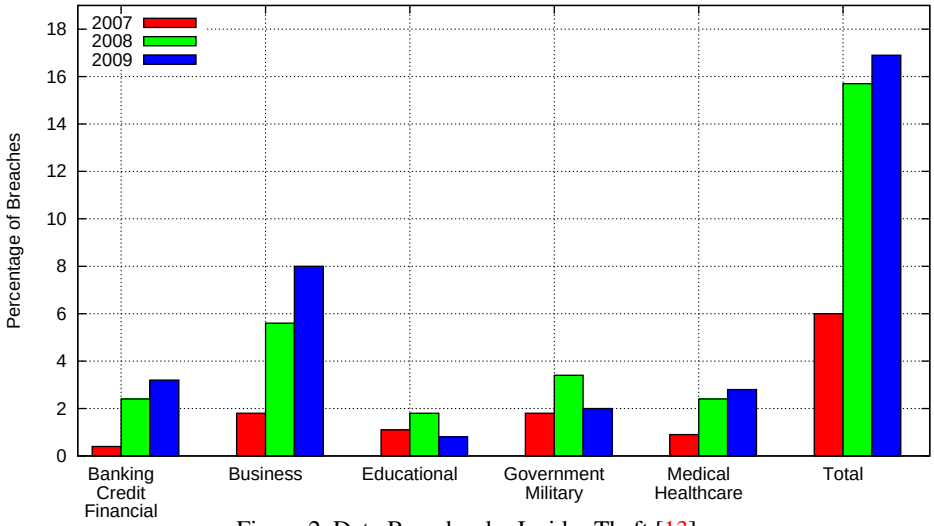

Figure 2. Data Breaches by Insider Theft [13]

Insider Theft (17%), Data on the Move (16%), and Accidental Exposure (12%). In Figure 2, we plot insider threat percentages by industry category and year. According to this data series, insiders are a significant and growing threat. In almost all cases, the exposed data was unprotected by strong technical means such as encryption. Subcontractors (secondary users) were responsible for about 10% of the reported breaches, and for about half of the total number of records released.

To summarise: businesses and governmental agencies should be concerned about insider threats, and also about disclosure threats from subcontractors and other semi-trusted second parties. Providing adequate security is difficult, even for a security-conscious organisation. This implies a market demand for cost-effective, easy to use, and trustworthy offerings of of "security as a service". The access control scheme proposed in this paper is a step in this direction.

2.2. *Principles of Watermarking*

We use definitions and notation from Collberg, Thomborson, and Zhu [14, 15, 16]. Let **P** denote the set of programs that are accepted by our watermarking system, **W** the set of watermarks for this system, and **K** the set of keys. A watermark is a message of bits expressed by 0 and 1 with a finite length ≥ 0.

Definition 1 (Embed Function)

$$\mathit{Embed} : P \times K \times W \rightarrow P' \tag{1}$$

The watermark-embedding function *Embed* is used to insert a watermark into a program. For $P \in \mathbf{P}$, $K \in \mathbf{K}$ and $W \in \mathbf{W}$, $P' = \mathit{Embed}(P, K, W)$ is called a watermarked program. The program P is called the original program corresponding to the watermarked program P'.

Definition 2 (Extract Function)

$$\mathit{Extract} : P' \times K \times W \rightarrow W \tag{2}$$

We use an *Extract* function to retrieve the watermark from a watermarked program, to verify the ownership of the software. Ideally, our extractor will have no false negatives, i.e. it will always retrieve a watermark that was embedded by our *Embed* function. This desiderata is formally expressed as follows.

DOI: 10.1002/spe

$$\forall P \in \mathbf{P},\, \forall K \in \mathbf{K},\, \forall W \in \mathbf{W} :\ \mathit{Extract}(\mathit{Embed}(P, K, W), K) = W$$

In practical settings, false negative (FN) errors arise for a variety of reasons, including imperfections in the watermark embedding and extracting algorithms, or adversarial modifications to a watermarked program P'. When reporting a false-negative rate (FNR), it is important to specify the test conditions. In particular, if a collection of watermarked programs is modified by a well-informed and well-resourced attacker, the observed FNR will be much higher than in tests where P' is randomly modified or is unchanged.

False positives (FP) are a second form of error, unavoidable in most practical settings. Occasionally a spurious watermark is extracted from an unwatermarked program, because of algorithmic imperfections or a clever attacker's ability to mimic our *Embed* function. A non-adversarial false positive rate (FPR) can be measured in a reproducible fashion, if the test conditions are carefully specified.

Adversarial FP and FN rates are highly variable, depending greatly on the skill and knowledge of the attackers. In practice, tiger teams composed of skilled hackers are used to assess the security of complex systems. These assessments do not produce accurate predictions of FPR and FNR for the system, but they usually uncover some vulnerabilities which can be incorporated into its risk assessment.

When severe vulnerabilities are discovered, a system may be abandoned, redesigned, or tolerated as-is, depending on the estimated cost-effectiveness of the mitigations for the vulnerabilities. For example, despite many costly mitigations over many years, the credit card system is still vulnerable to many types of abuse. The total cost of credit card fraud to U.S. issuers was $1.3 billion in 2007 [17]. That year, online merchants and consumers lost an estimated $15 billion to card-related frauds. This fraud rate – approximately 1% of the total value of credit card transactions – is apparently tolerable. Even higher error rates are tolerable in some applications, for example a false-negative rate of 10% in a fingerprint recognition system may be acceptable if it is used for logging into a laptop computer.

Practical watermarking processes exhibit a third type of error. A fail to mark error (FTM) arises when the output P' of the *Embed* function is not semantically equivalent to the unwatermarked input program P, or if the embedding process produces an error message indicating it is unable to mark P. Analogous errors are called "failures to enrol" in biometric systems, whenever a biometric is not captured successfully and recorded properly for future use.

In a watermarking system, as in a biometric system, it is unacceptable to damage the underlying object P. Accordingly, only conservative, semantically-safe, transformations should be used when watermarking a program, and the embedding software should be very thoroughly tested. With such precautions, watermark embedders will be as trustworthy as optimizing compilers, and the fail to mark rate (FTMR) of the watermark system will be determined almost completely by the limitations of the embedding process. A well-designed watermark embedder will produce an error message, rather than a buggy P' or one with an unreadable or incomplete watermark, whenever it is unable to find enough "safe places" in P to embed all of the bits of W.

In Table II, we summarise many of the published algorithms for software watermarking. The algorithms are distinguished, in this table, by the type of changes made to the program during the embedding process, by whether the *Extract* function is "blind", and by whether the *Extract* process requires the program to be executed during the extraction or recognition process. As indicated in the table, most published algorithms embed a watermark by extending the semantics of the program, generally by adding code (E3 in this table). The earliest watermarking methods renamed variables or reallocated registers (E1), or rearranged code blocks (E2).

In a non-blind watermarking algorithm (B1), the extractor compares the watermarked program to a copy of the unwatermarked original program. Non-blind algorithms are generally infeasible if a large number of different programs have been watermarked. Accordingly, most researchers have concentrated on developing blind algorithms (B2).

In a static watermarking algorithm (R1), the watermark can be retrieved by analysing the watermarked code. This is preferable for performance reasons, however security may suffer. If

DOI: 10.1002/spe

Table II. Summary of Watermarking Algorithms [18][19]

Algorithm	Embedding			Blindness		Recognizing		
	E1	E2	E3	B1	B2	R1	R2	R3
Register Allocation	√				√	√		
Davidson-Myhrvold		√		√		√		
Spread Spectrum			√	√		√		
Dummy Method			√		√	√		
Graph-Theoretic			√		√	√		
Opaque Predicates			√		√	√		
Abstract Interpretation			√		√	√		
Semi-Dynamic Multiple			√		√		√	
Collberg-Thomborson			√		√			√
Path-Based			√		√			√
Thread-based			√		√			√

Abbreviations: E1: Renaming, E2: Reordering, E3: Extend program semantics; B1: Non-blind, B2: Blind; R1: Static, R2: Semi-dynamic, R3: Dynamic.

the attacker can recognise the coding sequences that are indicative of a watermark, they may be able to modify or remove it. For this reason, static watermarking algorithms should employ a very large coding dictionary, or some other method of introducing a high degree of diversity in the watermarked code (such as a randomly-selected series of obfuscating transformations), so that pattern-matching attacks are infeasible. Robust error-correcting codes are also important, so that the watermark string is still recognisable if many of its bits have been modified by a partially-successful attack. The watermark may be encrypted, to make it more difficult for the attacker to determine whether a randomly-selected watermark extracting algorithm and key are the correct choices.

In a dynamic watermarking algorithm (R3), the watermark is extracted by examining the dynamic state (in registers, stack, heap, or memory) when the watermarked code is executed with a secret input. An intermediate approach, called semi-dynamic watermarking, hides a program in the watermarked code during the embedding process. The watermark is exhibited by this hidden program, whenever it is extracted and then executed. Semi-dynamic and dynamic watermarks rely in part, for their security, on a static analysis being unable to predict exactly what a program will do. Furthermore, some dynamic properties of the code, such as its data structures and the locking behaviour of its threads, are resistant to the standard techniques of code obfuscation – because safe obfuscators can only adjust code properties which can be statically analysed. Despite their security advantages, dynamic watermarks are infeasible in some applications, due to the greater computational resources required for a dynamic extraction in comparison to a static extraction.

Most academic research into software watermarking has been conducted on Java source or Java bytecode, due to its platform independence and its amenability to static analysis. However, the techniques listed in our table could be applied in any modern programming language.

3. SECONDARY USER THREATS IN COLLABORATIVE CLOUD COMPUTING

It has been claimed that "agent-based models enable richer descriptions and analysis techniques about Internet-based environments, especially ones involving intelligent agents" [20]. We have successfully used an agent-based modeling technique [21] to analyze steganography threats in VoIP [22]. In this section, we exploit this technique to analyze the threats to the software collaborations in the cloud.

We start by exploring the nature of collaboration for corporates, non-profits, and governmental agencies. The OCLC, a non-profit organisation representing 72,000 libraries in 170 countries and territories, describes the collaborative landscape as follows.

DOI: 10.1002/spe

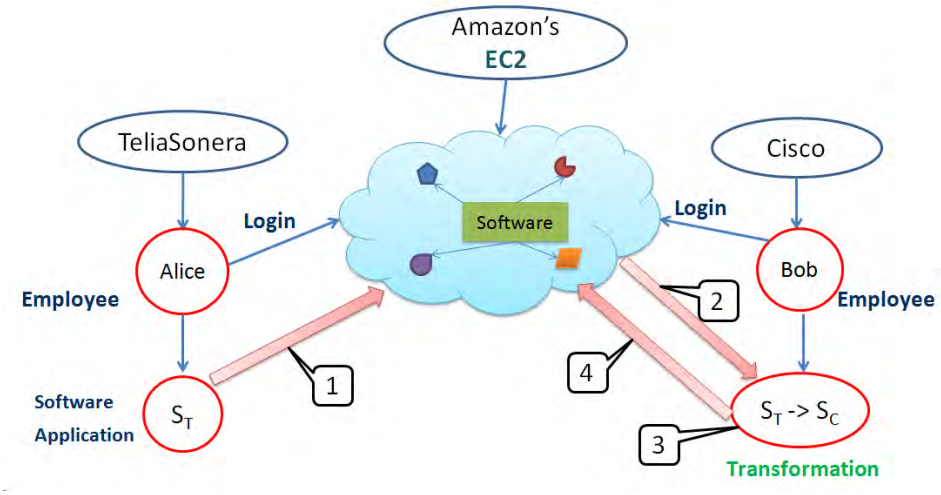

Figure 3. Secondary User Threat Model for Software in the Cloud

> From the perspective of a large institution (e.g., a university campus) with many units (e.g., libraries, archives, and museums), incorporating collaboration into the underlying work culture is foundational to realizing that institution's potential and achieving its mission... none of the collaboration contexts (local, group, or global) is inherently better than the other... Within any of these three contexts, the collaboration can be very shallow or very deep... In transformative collaborations, participants find efficiences that free up time and resources to focus on the things they do best [23].

We take, as a case in point, a March 2010 press release from Swedish telecom operator TeliaSonera and network equipment supplier Cisco. According to this press release, these companies are now collaborating on the provision of cloud computing services to business customers. They currently offer cloud-based conference and services to integrate video, telephony, chat and file sharing. They intend to expand the cooperation with more offerings in the future.

Technical details of this venture are not available to us, however for definiteness we assume that Cisco and TeliaSonera are planning to use some EC2 cloud services from Amazon [24]. In particular, we assume that Alice is a software developer at TeliaSonera, that Bob is authorised by Cisco to use Alice's software, and that Alice and Bob are communicating via EC2. The blue arcs in the diagram indicate a controlling relationship. For example, TeliaSonera controls Alice through her employment contract, and also (we presume) by rewarding and punishing some of her observed activities. Bob is under the control of TeliaSonera, but he is not under the direct control of Cisco nor of Amazon. Control is not absolute in our diagram: agents may perform activities without alerting their controller. Accordingly, Bob may be able to on-sell a copy of Alice's application for private gain. This threat is depicted in Figure 3, and consists of the four phases listed below.

- Phase 1: Alice logs into EC2, uploading a software application (confidential to TeliaSonera) to the cloud in order to cooperate with Cisco employee Bob.
- Phase 2: Bob logs into EC2, obtaining an authorised copy of Alice's application. Amazon is ultimately in control of all login sessions; formally, in our model, Bob's login session is an alias of an actor that is controlled by Amazon.
- Phase 3: Bob modifies the visible label on Alice's application, using obfuscation to transform the code so that a code comparison using `diff` would not reveal its origin. The new label on the application indicates that it belongs to Bob, in his private capacity as a "moonlighting" software developer. In a detailed model, Bob has two aliases: as an employee under the control

DOI: 10.1002/spe

of Cisco, and as a self-controlling moonlighter. We have suppressed aliases, for simplicity, when drawing Figure 3.
- Phase 4: Bob uploads the revised application to the cloud, in the hope of selling it to someone who would not be aware (or not care) that it actually belongs to TeliaSonera.

In our modelling technique, security goals are determined by human fears, and functional goals are determined by human desires. Any goal conflict is a threat to security or functionality [22]. In any insider threat, a single person has two aliases (e.g. "Bob the employee" and "Bob the moonlighter") with disparate goals. The security boundary for an insider threat is infeasible for anyone other than the threat agent to patrol completely, for it is essentially a mental firewall between two aliases (or roles) of the same person. Because Figure 3 depicts a threat from a secondary user, we can construct a security boundary between Alice and Bob by contrasting their desires and fears, and we need not consider Bob's desires and fears in his role as an employee of Cisco.

Alice, as a trusted employee for TeliaSonera

- Desire: Maintain information security for TeliaSonera, in part for financial advantage and also as a matter of professional pride.
- Fear: The application I wrote for TeliaSonera may be improperly published, which may cost me my job and make me feel that I haven't met my professional obligations.

Bob, as a threat agent in Cisco

- Desire: I want to steal valuable software applications from TeliaSonera, for financial advantage.
- Fear: My illegal behavior will be discovered, and I will be punished.

Access control in EC2 will effectively mitigate threats from outsiders, but it can not prevent Bob (as an authorised party from Cisco) from taking an illicit copy of Alice's application. If the access control provided by EC2, as configured by Alice, is not fine-grained, then many employees at Cisco will be able to access Alice's software. This generalised threat, from many agents at Cisco in addition to Bob, is mitigated by Cisco's information security policies and procedures. In particular, routine backups of Cisco's filesystems would, we presume, contain incriminating traces of Bob's nefarious activity. However these backups would be inspected only if someone at Cisco is, somehow, alerted to Bob's malfeasance.

To mitigate this secondary user threat, we must find an inexpensive, non-intrusive way to alert Cisco and TeliaSonera when Bob improperly publishes the application Alice supplied to him. Ideally, this alert should have no false-positives, i.e. it should raise no false alarms. It should have no false-negatives, i.e. it should raise an alarm even if Bob is clever about removing labels and obfuscating code. It should also be rapid, ideally raising an alarm so quickly that an immediate intervention (by Amazon) would effectively prevent Bob from on-selling the transformed code. In Section 5 of this paper, we will consider these technical desiderata in the context of the Jericho Forum's evaluation framework for Collaboration Oriented Architectures [6].

4. THE PROPOSED WATERMARKING-ENHANCED CLOUD

We now explore the prospect of applying a watermarking process in the cloud architecture, with the goal of mitigating the secondary-user threat identified in the previous section.

4.1. Architecture

Cloud computing is generally considered to have three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [3]. Alice and Bob, in our threat model, are using a filesystem provided by an IaaS layer in a cloud. We recommend that they

DOI: 10.1002/spe

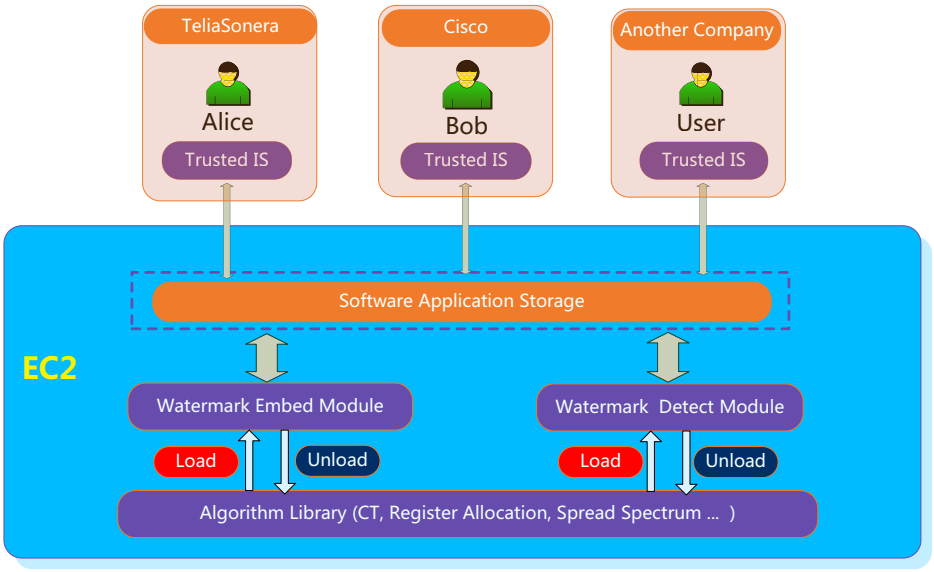

Figure 4. Watermarking Enhanced Cloud

shift to the watermark-enhanced SaaS depicted in Figure 4. For consistency with our prior model, we have assumed that Amazon's EC2 provides the underlying infrastructure. We then provide a watermarking service in a SaaS layer.

Our SaaS uses explicit (non-steganographic) security labels, in XACML, for most of its distributed digital-rights management services. However it also embeds watermarks, upon request from the uploader Alice, to mitigate the secondary-user threat from Bob. These watermarks must be robust enough to resist Bob's removal attempts. Furthermore, our SaaS inspects all uploaded files for watermarks. If Alice's watermark is detected in a file without an XACML label, or if her watermark string is found in a file that doesn't name her as an owner, then the SaaS provider will investigate the apparent misappropriation of Alice's software. Care must be taken, during the SaaS response to an alert, to consider the possibility of a false positive. The privacy and reputation of the suspected attacker cannot be compromised until the watermark reading is corroborated. However if Alice has only a few files bearing the same watermark, there will be only a few candidates to be inspected for similarity to the newly-uploaded code bearing her watermark but not her security label.

In our architecture, several components are highly trusted: the EC2 provider, the trusted IS infrastructure at the uploading enterprise (TeliaSonera), and the commmunication channels between EC2 and the collaborating enterprises. Cisco and Bob are partially trusted: they are assumed to be secure against outsider attack but are susceptible to secondary-user threats. The contractual terms of service between Cisco, TeliaSonera, and Amazon should reflect these levels of trust. By signing the contract, Cisco agrees to take responsibility if any file from TeliaSonera marked "for distribution only within Cisco" has been improperly distributed. Any alleged breach of contract will be investigated by trusted staff in all three companies, by examining logfiles and archival backups of corporate-controlled computers. The details of the response to an alleged breach is beyond the scope of this article. Instead, we concentrate on the difficult problem of detecting the breaches.

There are other, more complex, threats from secondary users than the one we study here. Of particular concern is that Bob might on-sell the modified software using some channel other than EC2, especially if he is aware that EC2 is embedding and detecting watermarks. We cannot hope to mitigate all important distribution threats from secondary users unless we enumerate all of the channels that Bob might use, and then find a way to limit his improper distributions on the most threatening of these channels. However if cloud computing becomes pervasive, then it would be

DOI: 10.1002/spe

in the financial interest of all reputable cloud service providers to collaborate with each other in their detections and responses to secondary user threats. Otherwise they will not be providing adequate information security to clients in disparate enterprises who are collaborating, and who have agreed to maintain the confidentiality and integrity of their sensitive collaborative exchanges. The contractual arrangement between Cisco and TeliaSonera should thus include a provision that neither should allow its employees to export collaboratively-shared files to any cloud which does not offer a compatible digital rights management system with a compatible watermark detector. Compliance with this agreement can be verified by security auditors. The secondary-user threat in our scenario is, in this respect, quite different to the threat of illicit distributions of MP3 files by consumers – because consumers, unlike Cisco and TeliaSonera in our scenario, have neither the resources nor the motivation to hire security auditors.

It is very important, in our architecture, to limit access to the watermark embedder. Bob cannot be allowed to embed arbitrary watermarks. Instead, the watermarks embedded by EC2 must indicate the employer of the person who requests a watermark. That is, we assume that TeliaSonera and Cisco both have secure processes for identity management, and for exporting their employees' identity credentials to EC2. If these processes are ever breached, then the secondary-user threat is a minor consideration; the primary problem, in such a scenario, is that these corporations are unable to reliably distinguish insiders from outsiders.

Some readers may be concerned about the high level of trust we are placing in EC2 and Amazon. TeliaSonera and Cisco would, we presume, not place this trust blindly. They might require Amazon to produce reports on its periodic security audits. They might, periodically, conspire against EC2 by enacting the Alice-Bob threat scenario, then see if an appropriate response is forthcoming from EC2. Finally, they might collaborate with many other potential corporate users of EC2 services, jointly selecting a security auditor which they mutually trust, and then requiring Amazon to open its EC2 facilities to inspection by this auditor.

Insider threats from within TeliaSonera are beyond the scope of this article.

We return now to our primary focus: the detection of an unauthorised distribution of software in a cloud by a secondary user. Figure 5 depicts the steps that should be taken by Alice and EC2, in order to detect a secondary-user distribution attempt by Bob.

1. Alice logs into EC2, uploading an access-controlled application P_L. The explicit label L on this application indicates that it was created by TeliaSonera and is authorised for distribution only to Bob of Cisco.
2. EC2 watermarks P_L, producing P'_L.
3. Bob logs into EC2, obtaining an authorised copy of P'_L.
4. Bob attacks P'_L, by extracting its code P', obfuscating it to Q', and affixing a new XACML label L' indicating that it was created by Bob of Moonlight Enterprises Ltd.
5. Bob uploads $Q'_{L'}$ to the cloud.
6. The watermark detector in the cloud examines $Q'_{L'}$, discovering a watermark indicating that its rightful owner is TeliaSonera. This watermark is inconsistent with the explicit label L', causing EC2 to commence an investigation of the provenance of $Q'_{L'}$. EC2 will inspect and run Q', to see if its code or functionality closely resemble any of its archival copies of code (including P_L) that it watermarked previously for TeliaSonera.

4.2. Watermarking Embedding and Detection

In this section we discuss some practical considerations in the design of our watermark. We also describe our prototype implementation using MapReduce.

As discussed in Section 2 and as depicted in Figure 6, there are two stages in the watermarking process: embedding and detection.

In our architecture, the watermarking key K must be known to EC2, but should remain a secret from the adversary Bob – otherwise the key provides no security advantage. The key cannot be specific to the program being watermarked, otherwise EC2 would not be able to retrieve a watermark from an arbitrary program of unknown or dubious provenance, such as $Q'_{L'}$ in our threat scenario.

DOI: 10.1002/spe

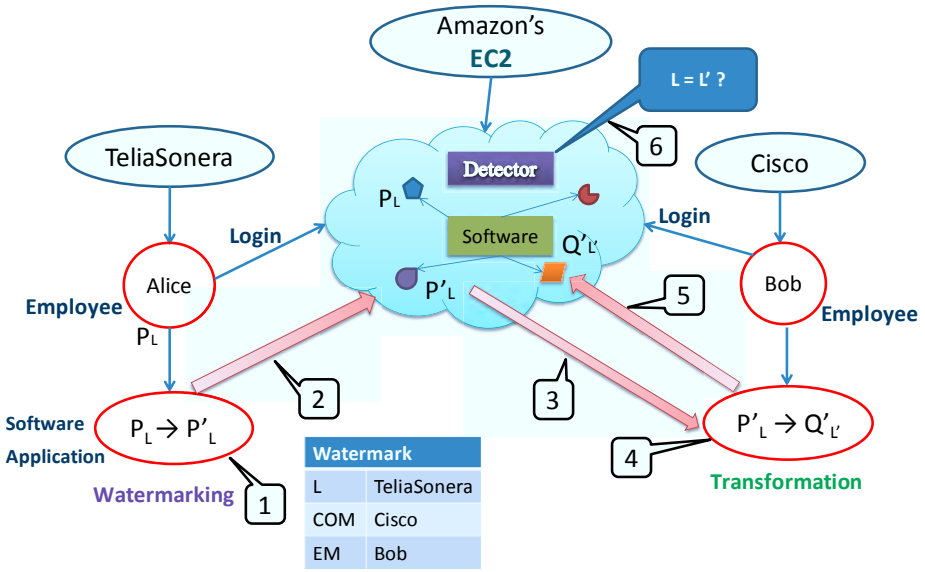

Figure 5. Watermarking Method for Software Protection in the Cloud

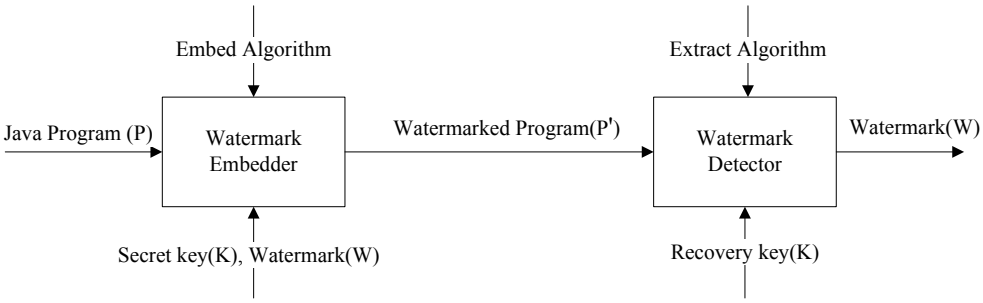

Figure 6. Watermark Embedding and Detection

This precludes the use of dynamic watermarks for which the keying is a secret input string for the program being watermarked.

The detailed design of W will vary, in practice, depending on the breadth of the distribution. However, the watermark string W must reveal the name of the company whose employee released it into the cloud: TeliaSonera in our example. Furthermore, to support an efficient investigation of an alleged infringement, watermarks should be diverse. If many software applications from a single company bear identical watermarks, then the investigator will have many previously-watermarked programs (P_L in our scenario) to compare with a program ($Q'_{L'}$) of dubious provenance. An appropriate level of diversity could be obtained by incorporating information from the explicit XACML label on the program, such as the name of the company, Cisco in our example, which is authorised to access the software. The name (Bob) of the specific person who is authorised to access the software would also be appropriate, in cases where the release is to a single person. However if many people in Cisco are authorised to access the software, then including all information from L in a watermark would be inappropriate, for it would allow two employees of Cisco to collude by comparing their differentially-watermarked codes. Such a differential analysis will, in general, reveal clues about the *Embed* process to a highly skilled attacker.

DOI: 10.1002/spe

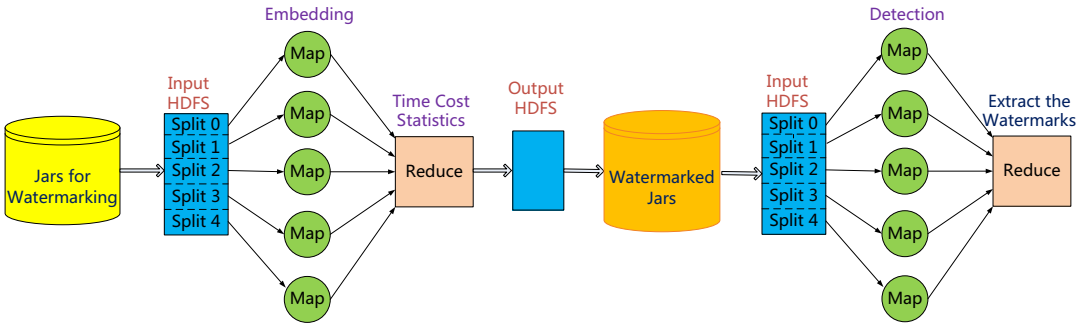

Figure 7. Watermarking Process in MapReduce

To mitigate the threat of a collusive attack, then, our modified EC2 will offer a watermarking service in which the same application will never be watermarked twice. Our watermarking service must retain archival copies of unwatermarked programs in any event, for their investigations, and duplicated requests can be detected efficiently by comparing hashes of the application code. Watermark strings should contain the name of the uploading company plus a serial number or date. They should be encrypted under a key known only to Amazon, so that the watermark is not susceptible to a known-plaintext attack by any attacker who doesn't know its decryption key.

Our modified EC2 will also offer a fingerprinting service. In this service, different watermarks are embedded for different authorised recipients. As with the watermarking service, the fingerprint string should be encrypted to mitigate known-plaintext attacks on the *Extract* function. The fingerprinting service will be more susceptible to collusive attacks than the watermarking service, but in the absence of successful collusion, the fingerprint will reveal the identity of the authorised recipient, thereby lessening the time and expense required in a forensic analysis to identify the guilty party.

Our prototype, described in the next section, fingerprints programs with the following information: the name U of the uploading company, the name E of the person who is authorised to retrieve the uploaded program, and the name C of the company who employs U.

4.3. Implementation in MapReduce

As shown in Figure 7, we designed and implemented our prototype using Hadoop, an open source Java codebase. Hadoop provides a programming model for cloud programming called MapReduce. In this model, large workload is divided into many small fragments. The fragments are then distributed to slave nodes in the cloud. Each slave process executes a Map procedure on a single fragment. The fragmentary results are then combined into a final result, using a Reduce() function.

Our pseudocode, displayed in this section, indicates how we used MapReduce to embed and extract watermarks for 100,000 JAR files. Others have used the cloud to watermark a million or more images but, to the best of our knowledge, our prototype is the first to watermark Java programs in the cloud.

Our prototype includes a graphical user interface, on which a user (Alice) can submit an application for watermarking. The interface allows her to set relative priorities for the Efficiency, Security, and Data-Rate of the watermark. Our back-end will select an appropriate watermarking method, given the user's priorities. The priority given to data-rate is relevant only when a user attempts to embed a long watermark in a small application. A watermarking method with an unbounded data-rate can embed arbitrarily-long watermarks in small applications, with low security. By contrast, a watermarking method with a very low data-rate can embed only a few bits of watermark per kilobyte of code. A screenshot of our prototype interface is shown on the left side of Figure 8. Other parts of the figure illustrate our hardware configuration (four slave PCs and a

DOI: 10.1002/spe

Procedure Embed.Map(f)

Input : f – name of a JAR to be watermarked; A, K – watermarking algorithm and key

```
1 P = getJar(HDFS, f) ;                  // Get the JAR from the Hadoop filesystem
2 FPerr = (Extract.A(P, K) ≠ ∅') ;                      // Test for false positives
3 W = "TeliaSonera.Cisco.Bob" ;        // Concatenate names to form the watermark
4 P' = Embed.A(P, K, W) ;                                  // Embed the watermark
5 FTMerr = (length(P') = 0) ;        // Was an error detected during watermarking?
6 putJar(HDFS, P') ;                 // Write the watermarked JAR to the Hadoop FS
7 If (FPerr)  putfile(HDFS, "f.falsepos", FPerr);
8 emit(FTMerr, FPerr) ;                    // Errors are summed in Embed.Reduce()
```

Function Embed.Reduce(X)

Input : X – array of values emitted by Embed.Map
Output : EmbeddingTime – total time for embedding;
totalFP, totalFTM – total number of false positives and fail-to-marks

```
1 totalFP = 0; totalFTM = 0 ;
2 for each x ∈ X do
3 |   totalFP += x.1; totalFTM += x.2 ;
4 end for
5 EmbeddingTime = getTimecost(Embed);
```

Procedure Extract.Map(f)

Input : f – name of watermarked JAR; A, K – watermarking algorithm and key

```
1 P' = getJar(HDFS, f) ;                 // Get the JAR from the Hadoop filesystem
2 W = Extract.A(P', K) ;                                  // Extract the watermark
3 FNerr = (W ≠ "TeliaSonera.Cisco.Bob") ;  // 1 if a false negative, otherwise 0
4 putfile(HDFS, "f.w", FNerr + "; " + W) ;                // Record details in HDFS
5 emit(FNerr) ;                          // FNerr is summed in Extract.Reduce()
```

Function Extract.Reduce(X)

Input : X – array of values emitted by Extract.Map
Output : ExtractionTime – total time for extracting; totalFN – total number of false negatives

```
1 totalFN = 0 ;
2 for each x ∈ X do
3 |   totalFN += x ;
4 end for
5 ExtractionTime = getTimecost(Extract) ;
```

master), the Hadoop Distributed File System (HDFS), diagnostic output channel, and our provisions for creating an audit trail on the watermarking process.

Our Hadoop cluster has five computational nodes. Each node has a Pentium (R) Dual-Core CPU E6300 running at 2.8GHz, 4 GB of memory, and an 800 GB hard drive. The five nodes all provide service as slave nodes, and a second process on one node is the master node. Each node runs Ubuntu 9.10, JDK version 1.6.0_12, and Hadoop version 0.20. In the Hadoop profile, we place a 2 GB limit on the virtual memory available to any sub-process. Together, these five nodes comprise our cloudlet.

A full-scale cloud will have much greater computational resources than our cloudlet. A full-scale may also have much larger speed-of-light latencies, and possibly some bandwidth bottlenecks or significant routing latencies, between the infrastructural units that are cooperating on a single MapReduce process. These "scaling errors" in our experimental apparatus are significant threats to the validity of our experimental findings. Despite these threats, we are confident that the general shape of our experimentally-derived performance curves, and of some of our experimentally-derived performance parameters, would be similar to the curves and parameters which could be measured,

DOI: 10.1002/spe

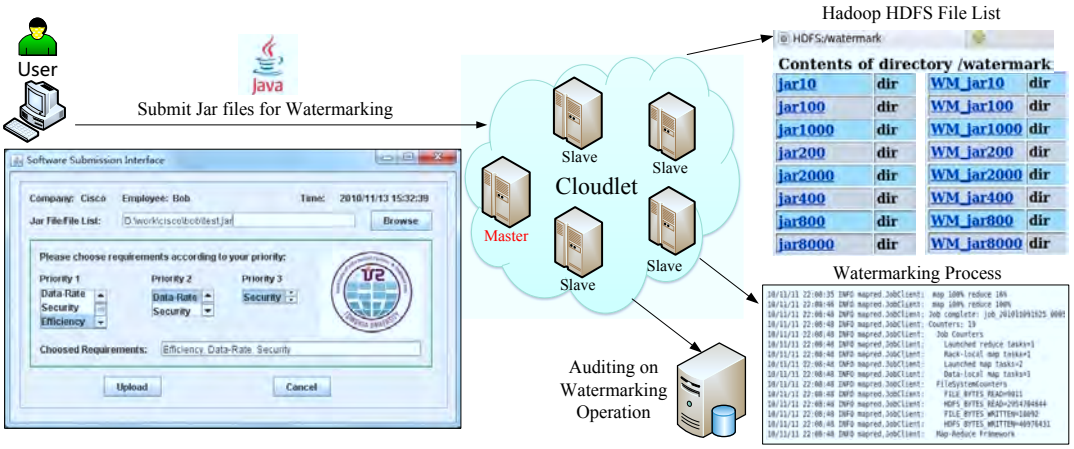

Figure 8. Prototype of Watermarking in the Cloud

by similar experimental techniques, on a full-scale cloud. In any event, we think it is generally advisable to build and test a prototype prior to building and testing a full-scale system.

The workload for our watermarking cloudlet is composed of 100,000 JAR files that we created from the following sources. We collected about 100 JARs from open-source developments at sourceforge. We collected 376 JARs from the plugins directory of IDE Eclipse. We wrote 20 JARs ourselves, including most of the ones (KMP, LIS, Huffman, Knapsack) we used for our evaluations of the time and space overheads of watermarking. Finally, we synthesized another 99,500 JARs from our first 500, by randomly adding and deleting classes from different JARs. The resulting workload is, undoubtedly, somewhat less diverse than a random selection of 100,000 JAR files from internet sources. However we believe it to be a plausible workload for testing the scalability of our watermarking system. Our smallest JAR is 20 KB, and our largest is 2 MB.

We used the Opaque Predicates watermarking algorithm GA1 [25] in our experimentation. According to Myles and Collberg [26], the resulting watermarks are robust to most, but not all, of the obfuscating transformations which Bob might attempt if he had access to the SandMark codebase. We will address the question of watermarking security more carefully below.

5. EVALUATION

Our watermarking system is intended to support collaboration between enterprises, as defined earlier in this article. Successful collaboration, at the enterprise level, requires support on many levels. At the managerial level, the person responsible for organising collaborative support involving computer systems is usually called a Chief Information Officer (CIO). A large corporation may also have a Chief Information Security Officer (CISO). A group of CIOs and CISOs, from prominent corporations such as Boeing, HP, and Standard Chartered Bank, founded the Jericho Forum at the offices of The Open Group in January 2004. Their goal was, and is, to define and promote solutions to the problems facing enterprises who are heavily sharing information across their corporate boundaries. In 2008, the Jericho Forum published the following definition of Collaboration Oriented Architectures (COA).

> COA-compliant information architectures enable enterprises that use them to operate in a secure and reliable manner in an environment of increasing information threat, and where it is the growing norm to interact without boundaries, irrespective of the location of the data or the number of collaborating parties. [27]

Below, we evaluate our watermarking system against the five desirable attributes of a COA.

DOI: 10.1002/spe

Usability. "Security measures are non-intrusive, are readily managed by the relevant governing enterprises, and are easily understood by the individual end user [28]." The users of our prototype had little difficulty operating our interface, however all were familiar with academic publications on software watermarking. Feedback from less expert users suggests that the data-rate concept can be difficult to understand, and that an even simpler interface would be preferable. Ideally, from the point of view of usability, our software watermarking system would be fully integrated into an organization's existing, general-purpose distributed rights management system.

To maximise both usability and effectivenss, the watermarking would not be a separate step requiring end-user interaction. Ideally watermarking and obfuscating transformations would occur at the time the code is compiled; and a visible access-control label would be attached in a post-compilation step before it is provided to either Alice or Bob. However such a high level of integration is infeasible for the foreseeable future, for it would require an extremely high level of integration of an organisation's code-production processes with its code-distribution processes, as well as extensive and expensive changes in the compiler itself.

Availability. "Information shared between collaborating organizations should not be rendered unavailable either by mistake or by an adversary. This implies that any 'at rest' encryption keys are escrowed, and that information is held in open-standard formats [28]." Our prototype exhibits four availability vulnerabilities.

1. The cloud may be unavailable to Alice when she attempts to upload her software.
2. A fail-to-mark (FTM) error during the watermarking step may introduce a serious bug, making Alice's software effectively unavailable to Bob.
3. The cloud may be unavailable to Bob when he attempts to download the software.
4. Finally, and most subtly, the watermarking service may be unavailable to Alice if the watermarking extractor falsely detects (in an FP error) the presence of someone else's watermark in her software.

Experimentation on our prototype cannot estimate the availability of a cloud-based service from Amazon or any other provider. In our experience, internet outages do occur at a noticeable frequency in corporate and university settings; and internet connectivity can be sporadic when travelling. So we would not expect 5-sigma (0.023%) availability, but we might enjoy 4-sigma (0.62%) availability for internet access to a cloud-based watermarking service from a reputable provider.

We didn't attempt to measure the rate at which our watermark embedder introduces bugs into program code. However we suspect it would be approximately 4-sigma, for we are using somewhat unstable middleware (Hadoop 0.20) for our filesystem and our MapReduce primitives. The Opaque Predicates watermark embedder needs only a near-trivial static analysis to find safe places to insert its known-true and known-false branches, so we would be surprised if it were introducing bugs in the programs it watermarks. In a production version of our watermarking system, the bug-introduction rate should be extremely small: 6-sigma or better.

We observed no false positives in our experimentation, however we did not measure under adversarial conditions. As discussed in the next section, the false positive rate must be extremely low, 6-sigma or better, in order for the system to be economically feasible. Such a low rate of false positives is easily obtained for the Opaque Predicates method, because Alice has only a very small chance of writing code that seems to contain a single bit of such a watermark. For watermarking methods where spurious watermark bit-detections are more likely, appending 19 bits of error-detecting code to the watermark will reduce its FPR to below the 6-sigma level of 0.0002%.

Summing these error rates, we estimate the availability of a production version of our system to be 4-sigma because of the somewhat-sporadic internet connectivity we have assumed for Alice and Bob. This compares favourably to the availability of an obvious alternative design. In this alternative, Alice uploads her application to a cloud-based computational server which provides Bob with execute-only privileges. Alice is thus offering her software to Bob as a SaaS cloud. In this alternative design, Alice is very well protected against the secondary-user threat of unauthorised distribution. However Bob is susceptible to internet outages whenever he tries to run Alice's application. In our

DOI: 10.1002/spe

design, Bob is only dependent on internet connectivity once: when he downloads Alice's software. Furthermore, if Alice's software requires very high bandwidth to any of Bob's I/O devices, such as graphics-display hardware, then its performance would be unacceptable as a cloud SaaS.

Efficiency/Performance. "Security measures should not greatly affect the latency, bandwidth, or total cost of data retrieval, storage, or transmission. This implies that collaborating partners must possess the means to rapidly access decryption keys for all data in their possession for which they continue to have access privileges, allowing rapid date retrievals and offline malware scans [28]."

Our design allows Bob to have full access to Alice's decrypted software, and the watermarked software is only slightly slower than the original. Further performance detail is provided in the next subsection.

Effectiveness. "COA-compliant architectures should provide an effective approach to organizing and controlling secure data transport and storage among a wide range of existing and future corporate information systems [28]."

Our design allows Bob to use any software provided by Alice, without imposing any special requirements on his platform. As noted above, if a high degree of security against the secondary-user threat is required, and if Alice's software does not required high bandwidth to Bob's I/O devices, then Alice should provide Bob with execute-only access to her software on a cloud-based server. However interorganizational collaborations are characterized by a high degree of trust. Collaborating organizations must trust each others' access control systems, before releasing their sensitive documents outside the corporate boundary. In such a trusting context, our watermarking design provides a method for verifying that trust has been well-placed in the past, as well as providing a way to mitigate the secondary-user threat.

As noted by Collberg and Myles [26], opaque predicate watermarks can be removed or distorted by some obfuscating transforms. If Bob is sufficiently well-informed about the watermarks currently being embedded by our system, if he has access to powerful deobfuscators, and if he is willing to debug any bugs introduced during deobfuscation, then he can take appropriate countermeasures. However our watermarking design will significantly raise the bar for secondary user attacks, because the attacker must dewatermark as well as adjust the visible label. Even skilled attackers may be deterred by the possibility that a watermarking system is using more than one watermarking method. Discovering and removing one watermark does not guarantee that all watermarks have been removed. By this argument, our system is very effective against adversaries who are unaware that the software is being watermarked, and also against adversaries who are worried that they may not be able to discover and remove all watermarks.

Agility. "COA-compliant architectures must take into account the dimensions of timeliness and flexibility, so as to enable development of business-driven enterprise architectures that are appropriately flexible and adaptable to facilitate changes in business operations with optimal rapidity and ease, with minimal disruption [28]."

In our performance tests, watermarking with opaque predicates requires a few seconds of CPU time. Our prototype indicates that the setup time for watermarking in the cloud is negligible. If Alice wants to watermark and distribute n applications within the next hour, she should use at least $n/1000$ compute servers.

Alice must spend a few seconds specifying the access control labels and watermarks on her software. Overall, our prototype suggests that watermarking should improve agility, by giving Alice (and her employer TeliaSonera) more confidence that her software will not be inappropriately distributed by her secondary user Bob of Cisco.

5.1. Watermarking Time Cost Evaluation

As shown in Figure 9, when watermarking 100 or fewer programs, using a single PC is faster than running a MapReduce watermarking process on our cloudlet. The cutover point is 200 programs

DOI: 10.1002/spe

Number of Jar files	Embedding Time Cost (s)		Extracting Time Cost (s)	
	Local PC	Cloudlet	Local PC	Cloudlet
1×10^1	3.4×10^1	5×10^1	1.36	2.3×10^1
1×10^2	3.01×10^2	2.83×10^2	1.4×10^1	2.6×10^1
2×10^2	6.58×10^2	3.69×10^2	2.7×10^1	2.8×10^1
1×10^3	3.81×10^3	7.88×10^2	1.58×10^2	3.3×10^1
1×10^4	3.93×10^4	5.33×10^3	1.47×10^3	3.51×10^2
1×10^5	4.11×10^5	8.32×10^4	1.69×10^4	3.18×10^3

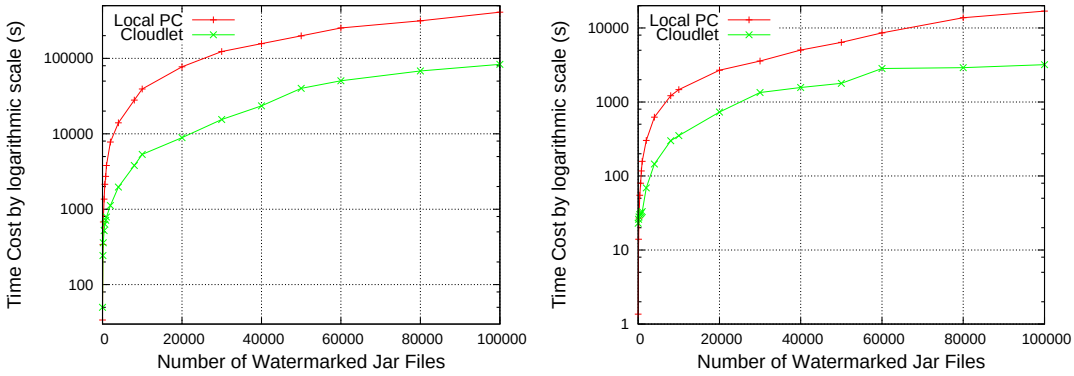

Figure 9. Time Cost of Watermarking in Local PC and Cloudlet. (a) Embedding. (b) Extracting.

when extracting watermarks. However requiring all watermarking to occur in the cloud, as in our design, will add at most a second or two to the completion time for even the smallest workloads. As might be expected, there is a five-fold speed advantage on our 5-PC cloudlet for large dataset. The speedup ratio is above 4x when embedding 1000 or more watermarks, or when extracting 10000 or more watermarks. Embedding a watermark using the Opaque Predicates method takes about 25x more time than extracting this watermark.

Figure 10 shows the time required to embed and extract a watermark for each file in our small (1000 JAR) dataset. These plots show the difference in completion timestamps for successive completions of watermarking tasks on each of our five slave PCs. Watermarking takes anywhere between 2.9 and 4.4 seconds per file, depending on the filesize. Extraction time is even more variable: between 10 and 170 msec. The 10 msec granularity in the extraction data is the length of one "tick" of the time-sharing clock in the operating system. The average time for embedding is 3.5 seconds, and the average time for extracting is 66 msec.

5.2. Overhead Evaluation

The Opaque Predicates watermarking process adds a few executable instructions to the Java software for each bit of watermark. This results in a modest increase in the size of the jarfile, as shown for a few files in Figure 11. For these files, the codesize never increases by more than 3.5%, even when a 64-bit watermark is embedded. When smaller watermarks are embedded, the codesize bloat is less, on average. This plot shows a significant granularity in the size-increase percentage, as a function of the watermark length, which defies a simple explanation. Because JAR files are compressed, the introduction of additional code bytes is likely, but not absolutely guaranteed, to increase the total size of the JAR.

The additional bytecodes introduced by an Opaque Predicate watermark are executed: they are not dead code. This is a significant security advantage, for it is impossible to remove a branch from live code safely, unless a static analysis reveals that the branch is always-true or always-false. However this security advantage comes at a cost in execution time. Our implementation of

DOI: 10.1002/spe

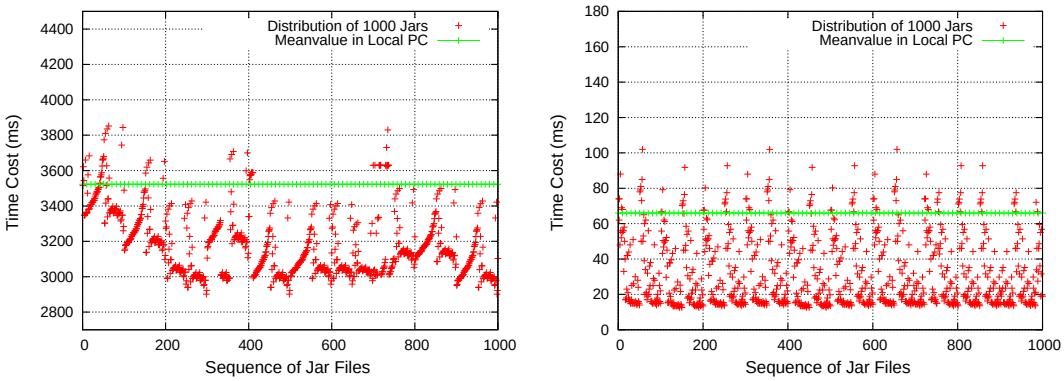

Figure 10. Distribution of 1000 Jars in Cloudlet. (a) Embedding. (b) Extracting.

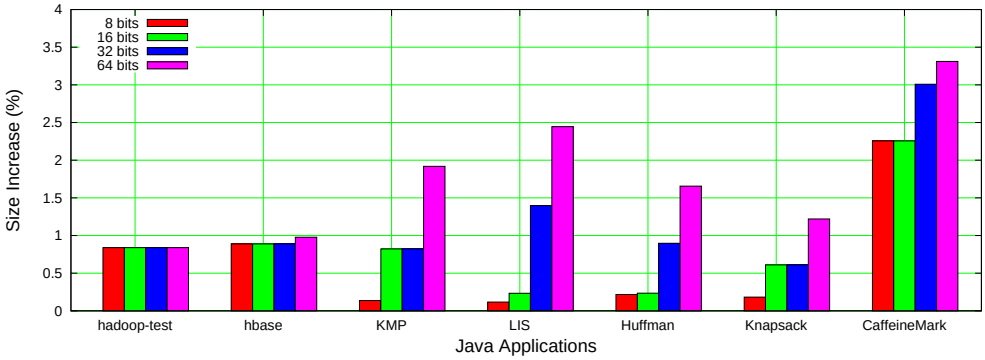

Figure 11. Evaluation Results of Size Increase

the Opaque Predicates watermark does not perform the (generally dynamic) analysis required to form an accurate estimate of conditional branch frequencies, nor does it even do the static analysis required to identify nests of for-loops. Accordingly, some of the inserted predicates will be in inner loops or other frequently-executed areas of code; and any insertions of code into such areas can significantly increase execution time in the worst case.

We evaluated runtime overheads in two different ways.

CaffeineMark. CaffeineMark is a series of microbenchmarks, each of which measures different aspects of the runtime performance of a Java interpreter and its execution environment. The CaffeineMark score, for each microbenchmark, is roughly the number of Java instructions executed per second. Our summary time metric, called "Overall" in Table III, is the geometric mean of the scores on the following six microbenchmarks (test items): Sieve, Loop, Logic, String, Float, Method, Overall. We repeated each measurement ten times, reporting only the averages here.

We define the runtime slowdown as follows:

$$RuntimeSlowdown = \frac{OriginalScore - WatermarkedScore}{OriginalScore} \times 100\% \quad (3)$$

As can be seen in Table III, we never observed a slowdown greater than 1.6%, and the overall slowdown was about 0.7% for 8-bit, 16-bit, 32-bit and 64-bit watermarks. We had initially expected to see the slowdown increasing with the size of the watermark. However the data shows no such increasing trend. We tentatively conclude that even the smallest (8-bit) watermarks are likely to insert an opaque predicate into the innermost loop of the loopnest in a CaffeineMark

DOI: 10.1002/spe

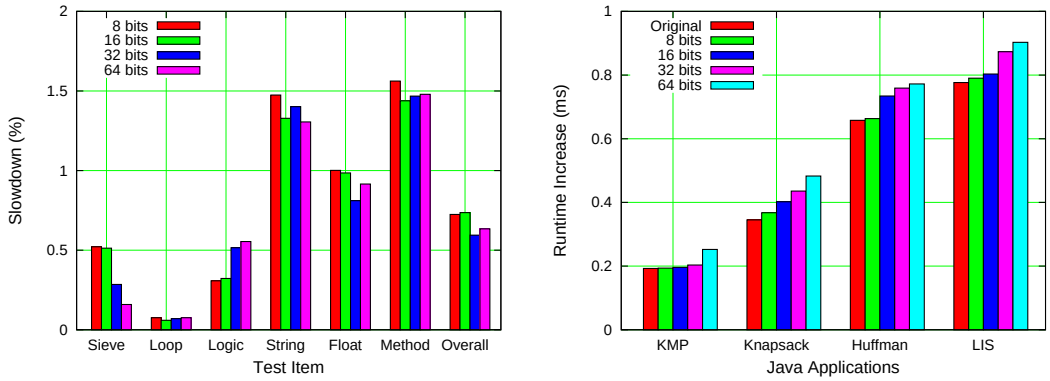

Figure 12. Runtime Overhead. (a) CaffeineMark. (b) Executable Java applications.

microbenchmark. The remainder of the watermark code insertions occur in less time-sensitive code, so the overall time performance is essentially invariant to the length of the watermark

Table III. Summary of the Watermarked CaffeineMark Scores

Item	**Score**	**8 bits**		**16 bits**		**32 bits**		**64 bits**	
		WS	**RS(%)**	**WS**	**RS(%)**	**WS**	**RS(%)**	**WS**	**RS(%)**
Sieve	54236	53899	0.6213585	53958	0.5125747	54190	0.1848145	54150	0.1585663
Loop	121815	121723	0.0755244	121758	0.0467923	121730	0.0697779	121723	0.0755244
Logic	68702	67735	1.4075282	68618	0.1222672	68348	0.5152688	68321	0.5545690
String	64577	63625	1.4742091	64131	1.2906484	63672	1.4014278	63734	1.3054183
Float	101267	99848	1.4012462	100269	0.9855135	100523	0.8113225	100349	0.9152725
Method	88646	87261	1.5623942	87371	1.4383052	87319	1.4676358	87335	1.4789161
Overall	80049	79476	0.727055	79542	0.7353620	79725	0.5947255	79543	0.6321131
Abbreviation: WS: Watermarked Score, RS: Runtime Slowdown									

We were pleasantly surprised to discover that the Opaque Predicates watermarking method, despite its lack of analysis to identify the most time-critical portions of a Java code, has only a very small effect on the runtime performance of the CaffeineMark microbenchmarks.

Executable Java applications. We conducted a second set of runtime overhead experiments, on four JAR files written over the past few years by members of our research team, to validate our results on the Caffeinemark JARs. The results are plotted on the righthand side of Figure 12. On these four files, as with the CaffeineMark microbenchmarks, the typical slowdown was a fraction of a percent. Most of these codes exhibit the linear dependence of slowdown on watermark length that we had originally expected. We suspect that this is because our codes do not spend most of their time executing a single loop nest, so that inserting the k-th watermark bit, for $k > 8$, has a significant chance of inserting code that will be frequently executed.

To summarise: our experience with Opaque Predicates watermarking suggests that it is very unlikely to significantly increase runtime. Even 64-bit watermarks, which insert 8 times as much code as 8-bit watermarks, caused no more than a 1.6% slowdown on any of the ten applications we tested. Usually the slowdown was less than 1%, which we think would be imperceptible to any user.

We recommend that any watermarking service agreement should disclose that the watermarking process is likely to cause a modest increase in the time and space required to execute the code.

DOI: 10.1002/spe

Table IV. Detection Performance with Diverse Watermark Sizes

Watermark Size	Fail to Mark (FTM)	False Negative	False Positive	Accuracy
8	0	0	0	100%
16	0	0	0	100%
32	0	0	0	100%
64	27	0	0	99.973%
128	156	0	0	99.844%
256	1335	0	0	98.665%

5.3. Watermarking Error Rates

We measured three error rates in the watermarking process: fail-to-mark errors (FTM), false negatives (FN), and false positives (FP). The test conditions were non-adversarial. A higher FN error rate would be observed in any situation where a clever adversary, such as Bob in our scenario, is employing an obfuscator in an attempt to prevent our watermarks from being detected. An adversary may also be able to create false-positives by transforming a program in such a way that it triggers a watermark detector.

Table IV contains a summary of our experimental results. The FTM is a count of the JARs which were not watermarked because our Opaque Predicates method could not find enough safe sites to insert watermarking code. Unsurprisingly, the FTM count increases sharply with the length of the watermark. The FTM rate becomes worse than 5-sigma, that is, greater than 0.23%, if the watermark is more than 32 bits in length. This is a disappointing result, for it implies that our prototype's method of formation of a watermark string by concatenating the 8-bit ASCII characters in a few names is infeasible.

In view of this experimental finding, we now recommend that the watermarking service should embed 32-bit hashes of strings rather than the strings themselves. This adjustment of our base design will greatly improve its FTM, and it will also improve its resilience to some adversarial attacks involving pattern recognition (because the watermarks will now resemble random bitstrings rather than ASCII text in a natural language).

Our measured false negative rate (FNR) and false positive rate (FPR) are both zero. The zero FPR implies that our system will generate no false alarms. The zero FNR implies that our system will prevent secondary-user fraud by any attacker who changes a visible label but who doesn't attempt to remove a software watermark. The Bob in our scenario will attempt to remove a watermark, but he can never be completely sure of his success. He will be deterred unless he is confident that his adversarial FNR is very high – well above 50% – because he will face a significant punishment if he fails to remove the watermark. If Bob is at all clever, he will realise that new types of watermarks may be introduced at any time, and that a single application may have many different watermarks. Accordingly, we recommend that the watermarking service have a provision for embedding multiple watermarks. If the security of the first watermarking process is ever compromised by a publication of de-watermarking software, then the second process should be activated.

6. SUMMARY AND CONCLUSION

We have examined the use of cloud computing for inter-organizational collaborations involving the transmission of copyright software from one party (Alice of TeliaSonera) to another (Bob of Cisco). We have identified a secondary-user threat to the security of these exchanges: Bob might redistribute the software for private gain or revenge. Because Bob, as an employee of Cisco, is authorised to use Alice's software (and he may even be authorised to edit it), it is difficult to prevent him from modifying its access-control labels. However, if Alice's software is robustly watermarked, Bob's unauthorised redistribution could – at least in principle – be detected.

Our contributions, in this article, were threefold.

DOI: 10.1002/spe

1. We articulated a secondary-user threat to collaborative sharing of copyright software. We argued that rights-management systems using the usual forms of protection (e.g. encryption, security labelling, and access control) are susceptible to this threat.
2. We presented a design for a cloud-based watermarking service. We argued that it will be in the interest of cloud service providers to provide such a service, and even to detect (and to report on) the watermarks embedded by other reputable cloud service providers. We evaluated our design against the criteria stated by the Jericho Forum [28].
3. We constructed a prototype of our cloud-based watermarking service, collecting and analysing experimental data on our practical experience with this prototype.

We conclude that our watermarking system, if refined to practice along the lines we suggest, would mitigate but not entirely nullify the threat of unauthorised distributions of copyright software by its secondary users.

Acknowledgments. This work is partially supported by the National Basic Research Program of China (No. 2009CB320706), the National Natural Science Foundation of China (No. 61073005, No. 90718010, No. 60803016), the National HeGaoJi Key Project (No. 2010ZX01042-002-002-01), and Tsinghua National Laboratory for Information Science and Technology (TNLIST) Cross-discipline Foundation.

REFERENCES

1. Business Software Alliance. Sixth annual BSA and IDC global software piracy study, May 2009.
2. Business Software Alliance. Seventh annual BSA and IDC global software piracy study, May 2010.
3. Cloud Security Alliance. Security guidance for critical areas of focus, Dec 2009.
4. Tharaud J, Wohlgemuth S, Echizen I, Sonehara N, Muller G, Lafourcade P. Privacy by data provenance with digital watermarking - a proof-of-concept implementation for medical services with electronic health records. *Sixth International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, 2010; 510–513, doi:10.1109/IIHMSP.2010.130.
5. Hwang K, Li D. Trusted cloud computing with secure resources and data coloring. *IEEE Internet Computing* Sept-Oct 2010; **14**(5):14–22, doi:10.1109/MIC.2010.86.
6. Jericho Forum. *Collaboration Oriented Architecture, Version 2.0* Nov 2008.
7. Armbrust M, et al. Above the Clouds: A Berkeley view of cloud computing. *Technical Report UCB/EECS-2009-28*, EECS Department, University of California, Berkeley Feb 2009.
8. Gens F. IT cloud services user survey, part 2: Top benefits and challenges, 2008.
9. Lian S, Zhang Y, Lian S, Zhang Y. *Handbook of Research on Secure Multimedia Distribution*. Information Science Reference: Hershey, PA, 2009, doi:10.4018/978-1-60566-262-6.
10. Lian S, Kanellopoulos D, Ruffo G. Recent advances in multimedia information system security. *Informatica (Slovenia)* 2009; **33**(1):3–24.
11. Hwang K, Kulkareni S, Hu Y. Cloud security with virtualized defense and reputation-based trust mangement. *Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC '09)*, 2009; 717 –722, doi:10.1109/DASC.2009.149.
12. Rindfleisch TC. Privacy, information technology, and health care. *Communications of the ACM* Aug 1997; **40**(8):93–100.
13. Identity Theft Resource Center. Data breaches, 2007-2009.
14. Collberg C, Thomborson C. Software watermarking: Models and dynamic embeddings. *POPL*, 1999; 311–324.
15. Collberg C, Thomborson C. Watermarking, tamper-proofing, and obfuscation — Tools for software protection. *IEEE Trans. Software Eng.* 2002; **28**(8):735–746.
16. Zhu W, Thomborson C. Recognition in software watermarking. *First ACM International Workshop on Content Protection and Security (MCPS 06)*, ACM: New York, NY, USA, 2006; 29–36, doi:http://doi.acm.org/10.1145/1178766.1178776.
17. Paterson K. *Credit Card Issuer Fraud Management*. Mercator Advisory Group Dec 2008.
18. Zhu W, Thomborson C, Wang FY. A survey of software watermarking. *ISI*, 2005; 454–458.
19. Collberg C, Nagra J. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.
20. Liu L, Yu ESK, Mylopoulos J. Security and privacy requirements analysis within a social setting. *Eleventh IEEE International Conference on Requirements Engineering (RE 2003)*, IEEE Computer Society, 2003; 151–161.
21. Thomborson C. A framework for system security. *Handbook of Information and Communication Security*, Stamp M, Stavroulakis P (eds.). Springer, 2010; 3–20, doi:10.1007/978-3-642-04117-4_1.
22. Yu Z, Thomborson C, Wang C, Fu J, Wang J. A security model for VoIP steganography. *Multimedia Information Networking and Security, 2009.*, vol. 1, 2009; 35 –40, doi:10.1109/MINES.2009.227.
23. Waibel G. Collaboration contexts: Framing local, group and global solutions, 2010.
24. Amazon. *Introduction to Elastic MapReduce*. API version 2009-03-31 edn. 2009.

25. Arboit G. A method for watermarking JAVA programs via opaque predicates. *Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.
26. Myles G, Collberg C. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research* 2006; **6**(2):155–171.
27. Jericho Forum. *Position Paper – Collaboration Oriented Architectures, Version 2.0* Nov 2008.
28. Jericho Forum. *Position Paper – COA Framework* Nov 2008.

DOI: 10.1002/spe