Protecting Document Provenance Against Exposure Events

ABSTRACT

Existing provenance management systems collect only *editing events*, ignoring *exposure events* (such as exporting a document). They also do not protect against tampering of the code and data of the provenance management system itself. These systems are therefore susceptible to insider attacks. We present the design of Haathi—a provenance management system that also captures provenance exposure events, uses text watermarking techniques to trace users who leak documents out of the system, and employs software protection techniques to protect against tampering.

Our design allows multiple text watermarking algorithms to be used simultaneously, for higher resilience to removal. Marks are position independent allowing text to be moved around without destroying the mark. All watermarking bits appear random, preventing collusive attacks.

1. INTRODUCTION

The *digital provenance* (DP) of a digital object gives a history of its creation, update, and access. It thus provides meta-level information of the sequence of events that lead up to the current version of the object, as well as its chain of custody. In particular, the meta-information describes the *what*, *where*, *when*, *how*, *who*, *which*, and *why* of the document's history [29]. "What" is the sequence of events that affect the data object. "When" and "where" record the time and location of a provenance event, respectively. "How" records the actions that lead up to an event. "Who" and "which" record the entities (individuals and organizations) effecting the event and the (software) tools used in the events, respectively. "Why", finally, describes the decision rationale behind the event.

1.1 Principals

Conceptually, the principals of a provenance-enabled system are *users* who create and modify digital objects, *auditors* who query the provenance of an object, and *validators* who check the validity of an object's provenance. This is illustrated in Figure 1. In practice, the same person can serve in different roles at different times, and auditors and validators can be automatic services rather than individuals. Users employ *Provenance Enabled Tools* (**PET**s), to create and modify objects and auditors use *Provenance Query Tools* (**PQT**s) to access DP information in order to learn about an object's history. Also part of a provenance-enabled system is a storage sub-system for provenance and the digital objects themselves.

1.2 Contributions

In this paper we present the design of a secure provenance collection system called *Haathi*[1]. Figure 2 shows the overall design. Haathi is targeted to office documents, such as text documents in word-processors, spreadsheets, drawings, and presentations. These types of documents are ubiquitous in the world of business, academia, and government, and being able to securely collect and manage their provenance is essential.

Specifically, the current version of Haathi implements a **PET** by extending OpenOffice's *Writer* tool. Haathi collects extremely fine-grained provenance, down to editing events such as adding and deleting individual characters and exposure events such as export and printing.

The design and implementation of Haathi simultaneously addresses these closely interlinked topics:

- to mitigate leakage of documents out of the system we present novel techniques for *robust, continuous marking, collusion-free, text fingerprinting*;
- to allow for efficient storage and access of fine-grained and potentially large provenance data our system supports *anonymous storage on untrusted storage servers*;
- to ensure end-to-end integrity of provenance data we investigate novel techniques for *continuously updatable software tamperproofing.*

1.3 Security and Usability Goals

Provenance-enabled document-processing systems have multiple goals related to the security and privacy of the provenance and the documents themselves, as well as usability goals related to functionality, reliability, and efficiency. We list these below.

The system should protect the **confidentiality of document contents and provenance** information from outside attackers, malicious insiders, unauthorized users, stor-

[1]The system can be downloaded from `http://haathi.cs.arizona.edu`. Haathi is Hindi for "elephant," the animal which, like our secure provenance system, never forgets.

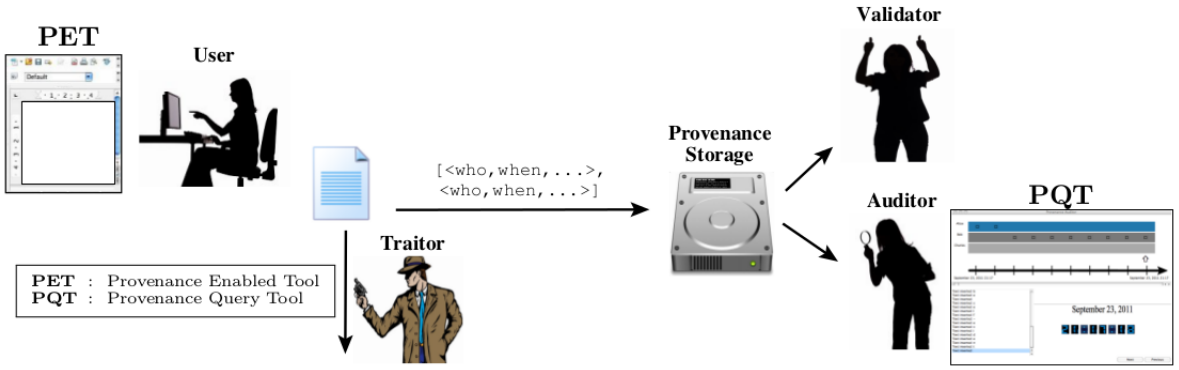


Figure 1: Principals and tools in a provenance-enabled system.

age servers, and auditors. The system must provide **access control** of documents and provenance: the system must only allow authorized users to modify documents, and only allow authorized auditors to access provenance data. Since any principal can be malicious, the system must provide **end-to-end provenance integrity**, such that any point on the chain from provenance collection to storage to auditing is protected. The system must collect **comprehensive provenance metadata**. This includes *access* events (a user requesting a document), *editing* events (a user modifying a document), *disclosure* events (a traitor leaking a document), as well as *auditing* events (an auditor requesting a document's provenance, also called a *meta-provenance* event). The system must provide **usable auditing facilities**, by keeping comprehensive and fine-grained provenance metadata since it cannot be known *a priori* which historical information an auditor might need. The system must be **reliant and scalable**, since, with fine-grained provenance collection, data can accumulate rapidly, and hence store and retrieve operations can become bottleneck operations. The system must allow the user to interact with their documents using **non-provenance enabled tools**, such as email clients. Accurate metadata cannot be collected if a document escapes the provenance system, which means that the provenance system must provide mechanisms to trace the source of such leaked information.

1.4 Insider Threat Model

The creators of an artifact are often the ones most motivated to mislead others about its history. Therefore, in contrast to previous work, we will assume malicious users who attempt to change *who*-provenance to implicate another user, *when*-provenance to hide modification time, or modify the document without this being reflected in the *what*-provenance.

Also in contrast to previous work, we consider document *leakage*. This occurs whenever the user prints a document, emails it, takes a screen shot of it, exports it to a different format, or copies information into or out of it, if the provenance is not augmented with the corresponding change-of-custody information.

Also in contrast to previous systems which have been coarse-grained, collecting information at the level of I/O-operations, ours collects provenance at the level of individual characters. This avoids attacks where the time when a small but significant modification ("friend" into the antonym "fiend") was made is not accurately reflected in the provenance.

2. RELATED WORK

Faced with the possibility of insider attacks, secure provenance systems need cryptographic techniques to create unforgeable metadata chains, steganographic techniques to trace leaked documents, and software protection techniques to ensure the integrity of code and data structures. We discuss prior work in these areas next.

2.1 Provenance Collection and Storage

Sprov [23] is an application-layer C library that replaces `stdio.h` in a Unix application. The library captures all I/O write requests and appends them to the provenance chain. One or more validators (called *auditors*) can verify the integrity of provenance chains. Sprov uses encryption, signatures, and checksums to ensure that adversaries cannot forge provenance chains (by inserting or deleting provenance records) that match an illicitly constructed document. Haathi incorporates these techniques to ensure the integrity of event chains.

A malicious Sprov user can simply tamper with the library code itself in order to circumvent the provenance collection. Thus, Sprov is not secure against insider attacks. Operations such as printing or exporting of a document are not tracked. Thus, Sprov does not support traitor-tracing of documents that have escaped the system. Sprov's provenance collection is coarse grained (raw I/O events) and thus information (such as time) about individual edit operations is not tracked.

PASS [27], like Sprov, collects I/O events as provenance, but does so at the Linux kernel level. Also like Sprov, PASS is susceptible to attacks by malicious users: no protection of the PASS kernel extension itself is provided, allowing adversaries to circumvent it by, for example, simply loading different kernel modules.

2.2 Traitor Tracing

Once a digital object escapes our provenance system (through printing, export to a different format, etc.), any further movements or edit operations will not be captured in our chain of provenance events. In this paper, we explore the idea of *continuous document fingerprinting* to mark docu-

ments with the identity of the user editing it, allowing for leaked documents to be traced back to their owners. Our method improves on the robustness of current techniques by allowing multiple simultaneous fingerprint carriers. In contrast to previous techniques we are also resistant to *collusive attacks* [20]; i.e. computing the difference between two closely related, and differently fingerprinted, versions of the same document does not reveal the location of marks.

Common ideas for document fingerprinting include embedding identifying data in character-, word- or line-spacing [7]; in the luminance [6,31,32], color [19], or shape of characters [30]; in synonyms or grammatical structures [2,3]; or in visually imperceptible dots [25]. Algorithms trade-off between robustness, embedding rate, and unobtrusiveness. For example, embedding in grammatical structures is robust to many attacks, but has a low embedding rate. Robustness also depends on the document format: line-spacing in PDF, PostScript, or XML [8] is more easily manipulated than in a bitmap, such as JPG. Robustness, embedding rate, and security can be improved by embedding multiple copies of a mark, by compressing and encrypting marks prior to embedding, and by error-correction techniques [11].

2.3 Software Protection

If a user is in complete control over a device and the software it contains he can reverse engineer and tamper with it at will. *Software protection* [16] techniques for protecting against such attacks include code obfuscation [4], tamperproofing [10,24], and software watermarking [13]. It is expected that, given enough time and effort, software protection defenses will be breached. In this paper we propose using *continuously updatable software protection* [14] to extend the lifetime of the protection.

2.4 Workflow Provenance

In *Scientific Workflow Provenance* [1,18,21,26,28], records are kept with detailed information about parameter settings and the sequence of steps that produced a particular data item. The provenance can later be queried to evaluate the validity and reliability of the data. The insider scenario investigated in this paper applies to scientific provenance as well: a dishonest researcher who fabricates or falsifies data or results will want to cover his or her tracks by manipulating the metadata to conform to the doctored data.

3. Haathi: SYSTEM DESIGN

Figure 2 illustrates our architecture for a secure provenance-enabled document management system. Users employ **PET**s to create and edit documents and auditors employ **PQT**s to query the resulting provenance. The *Trusted Authentication Server* (**TAS**) is operated by the users' organization, and is in charge of maintaining user identities and validating the provenance information. A **USS** is an untrusted storage server for provenance of documents. In practice, the **TAS** and the **USS** may be one and the same, but our design allows for expensive operations (such as storage of fine-grained, and hence large, provenance) to be outsourced to, say, honest-but-curious cloud storage providers. **PIT**s are tools outside the provenance system with which the user wants to exchange document data. The final component is the *Secure Provenance Library* (**SPL**) which is integrated into **PET**s and **PQT**s to collect provenance in a manner that guarantees authenticity and integrity.

Figure 3 shows how the OpenOffice Writer application has been turned into a **PET** by including our **SPL**. The **PET** maintains a copy of the document and the *local provenance store* (the provenance records of the document), until these are committed to the **USS**. The **SPL** monitors user actions and generates the corresponding provenance records into the local provenance store. A typical provenance record will contain the user identifier, the current time, the current geo-location, the operation performed, and so on. Thus, in addition to the information it keeps internally, **SPL** has to acquire some environmental information (`currentTime`, `currentLocation`) from the operating system. On a save, **SPL** packages up the provenance records and the new document version, encrypts and signs them, and sends them to the **USS**.

3.1 Design Challenges

In order to mitigate insider attacks in provenance-enabled systems we must address two fundamental challenges. The first challenge is to be able to trace documents that have escaped our system. For example, a malicious user may export the document being edited into a PDF or take a printout and leak it. Though proper provenance will be collected, it may not be sufficient if printouts of the same document were taken by multiple users at the same time. If any of these users decide to leak the document, we need to be able to trace the document back to the offending party. We use text watermarking techniques to fingerprint office documents with the identity of users. However, current techniques are not sufficient for our purposes since a) they target static documents (our office documents are continuously changing as the user performs edit operations), b) they are not resistant to collusive attacks, and c) they are not robust against many simple transformations. To overcome these short comings, we develop a watermarking framework that watermarks the document continuously and is robust to various kinds of attacks.

Our design incorporates the *Sprov* [23] hash chain scheme for protecting the integrity of provenance records. Unfortunately, without protecting the integrity of the *code and data* that computes these chains, the provenance remains insecure. Thus, a second challenge is to ensure the integrity of the entire provenance data processing chain. In practice this means that we must develop techniques for protecting the **SPL**'s code and data structures from modification, such that the path from where an event is first collected (in our case, where the user enters data into an office document), to where it is processed (in our case, where provenance events are encrypted and chained together using checksums), to where it is stored (in our case, on untrusted storage servers) is safe from tampering.

4. EXPOSURE EVENTS

Figure 4 shows the possible ways a document could escape the system. In either of these cases, the **PET** will add a record $\langle \texttt{User}_1, \textbf{export}, \textbf{D} \rangle$ to the provenance store. While nothing further can be said about subsequent whereabouts of **D**, at the very least we know from the provenance that at a particular time a particular document version was exported by a particular user. A malicious insider could circumvent the collection of export provenance events by

1. bypassing the tamperproofing that protects the **SPL**

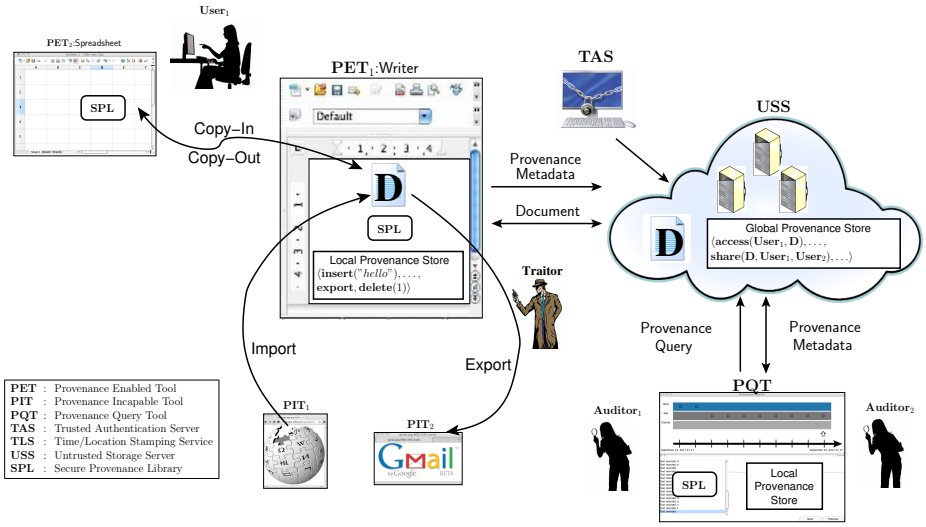


Figure 2: Overview of the secure digital provenance system.

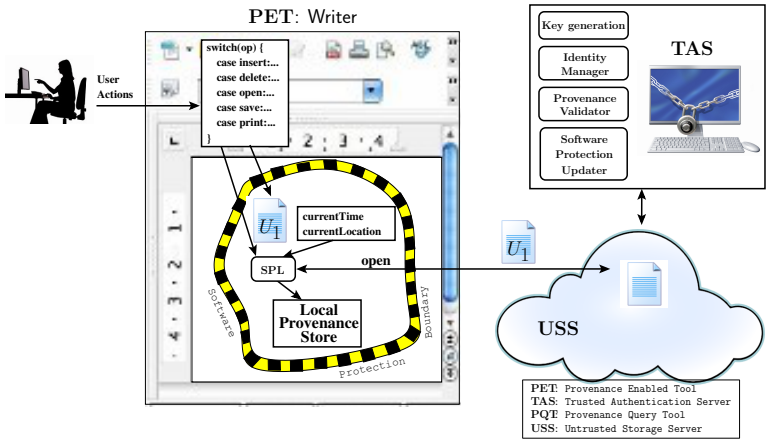


Figure 3: Overview of the **PET** and its interaction with the **TAS** and **USS**s.

(such protection can never be absolute);
2. capturing the screen of the **PET** using the operating system's built-in screenshot facility;
3. capturing the screen using an external camera;
4. manually transcribing the document (a laborious attack for which there is no defense).

We will use *traitor-tracing* techniques to mitigate these threats. These mechanisms will allow tracing the individual malicious user when an exposed document is detected. The idea is to *mark* the document with an identifier unique to each user. Such marks are known as *fingerprints* or *watermarks*, and there exists many marking algorithms for different kinds of digital objects, such as images [17], video, audio, text [2], and software [16]. Text watermarking algorithms are described in Section 2.2. The fingerprint allows an exposed document to be traced back to the client who leaked it.

Well-designed fingerprints should be resilient to document modifications and to collusion between clients. While collusive attacks have been studied extensively for image, audio, and video fingerprinting, we have been able to find no work on collusive-resistant text fingerprinting. Let M and N be malicious users, V a victim user, and D_U the version of document D held by user U. There are several related attacks:

self-similarity attack: M creates a version D'_M of D_M by making minor edits, compares D_M to D'_M to determine the location of the fingerprint marks, removes them, creating a clean version of D.

collusive attack: M and N compare D_M and D_N to locate the marks, remove them, creating a clean document.

spoofing attack: M gains access to V's version D_V of document D, compares D_V to D_M to locate and extract V's marks, and creates a new document into which V's marks are inserted.

While there has been much prior work on traitor tracing [12] through fingerprinting, our scenario is very different from what has been considered in the past and requires novel solutions. In our scenario, fingerprinting has to be performed by the **PET**, which is under the insider's control. This is different from standard traitor tracing scenar-

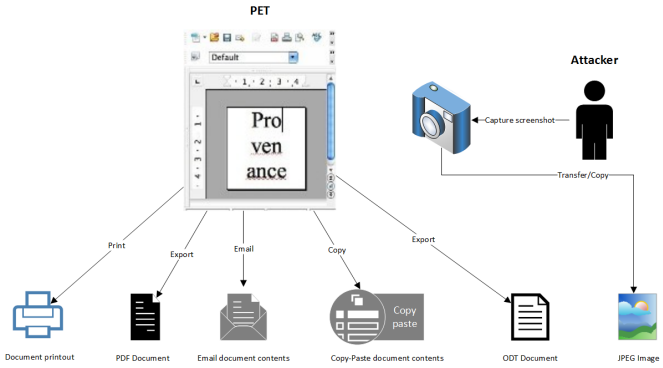


Figure 4: Possible threats due to exposure events.

ios where a trusted party marks the object before it is distributed to an untrusted party. In Section 5 we will discuss integrity protection of the **PET** code.

In the remainder of this section, we describe a *generic, continuous marking, collusion-free* text marking algorithm. *Generic*, means that to strengthen marks against a variety of attacks, rather than focusing on a single embedding carrier (such as word-spacing), we allow any combination of carriers. *Continuous marking* means that, as the user is typing and editing the document, the view that is presented on the screen is continuously updated, ensuring that screenshots will always be marked. *Collusion-free* means that every single marking bit, from an attacker's point of view, is random, and gives away no information.

4.1 Algorithmic Overview

To mark the text in the current viewport, a sequence of random 64-bit watermark numbers is generated, using a secure PRNG seeded with the user's ID and the current document version number. Next a 64-bit window is slid over the "embeddable bit-positions" in the viewport. These embeddable bits could come from any combination of watermark carriers such as luminance, color, spacing, character shapes, synonyms, etc. (see Section 2.2). At each 64th bit position we flip a biased coin and either embed the next watermark number generated by the PRNG (representing the identity of the user) or 64 bits of random chaff.

The embedding process is illustrated in Figure 5 where segment 1 contains chaff and segments 2 and 3 both contain random numbers identifying the current user. Each rectangular block represents one bit of embeddable watermark information. While the number of embeddable bits per character will depend on the chosen carrier (luminance, color, etc.), in the figure we assume each character to have a capacity of 8 watermark bits.

As a result of this marking process, a) multiple fingerprints are embedded in every section of the document allowing us to trace leakage of parts of documents; b) every carrier bit is random and collusive attacks are prevented; and c) multiple simultaneous de-watermarking transformations are necessary to perturb all watermark carriers, and robustness is improved.

The watermark engine is a 3-tuple $(\mathcal{E}, \mathcal{D}, M)$ where $M = \{M_1, \ldots, M_n\}$ is the set of available watermarking schemes, and $\mathcal{E}_{\{M_i,\ldots\}}$ and $\mathcal{D}_{\{M_i,\ldots\}}$, respectively, represent the watermark embedding and extraction components instantiated with a given subset of the watermarking schemes.

In the remainder of this section we will use the following notation (we will drop the i subscript when there is no risk of confusion):

- Let $\{D_1, \ldots, D_k\}$ be the set of documents in the system.
- Let $\{U_1, \ldots, U_n\}$ be the set of users who have access to a particular document D_i, where $1 \leq i \leq k$. Here n is a relatively small number (for example, $n \leq 32$).
- Let D_i^v be version v of document D_i.
- Let $\text{AES}(D_i)$ be a 128 bit AES key for document D_i.
- Let $\text{ID}(D_i)$ be the unique identifier for document D_i.
- Let $\text{PK}(U_i)$ be the private RSA key of user U_i.
- Let S_i^v be the unique seed generated for version v of document D_i.
- Let $D_i'^v$ be the watermarked document for D_i^v

4.2 Watermark Embedding

When a user U opens a document D^v for modification, a unique seed $S = \text{SHA256}(\text{PK}(U)||\text{ID}(D)||v)$ is generated. A PRNG is initialized with this seed and $\text{AES}(D)$ to produce a sequence of non-repeating random numbers unique to this user and document version. These random numbers are embedded as fingerprints in D. The watermarking scheme used to perform the embedding is randomly selected at the beginning of every 64 bit block. Either a fingerprint generated by the PRNG or random chaff is embedded in each block. Continuous modification of the document by the user results in live watermarking of the document.

Figure 6 shows the watermark embedding algorithm. Input to the algorithm is a version D_i^v of document D_i, consisting of a sequence of paragraphs, each paragraph a sequence of lines, each line a sequence of words, and each word a sequence of characters. From these attributes of the document a sequence of k embeddable bits $\langle b_0, b_1, \ldots, b_{k-1} \rangle$ is obtained. The watermark is embedded into these bits.

4.3 Watermark Extraction

The watermark extraction component takes a watermarked document as input — in the form of a PDF file, a structured document such as an Open Office file, or as a scanned copy (jpg, png, etc.) of the document — as well as the same subset of the watermark schemes used during embedding. As the number of users and documents produced by a particular organization is expected to be fairly small, an

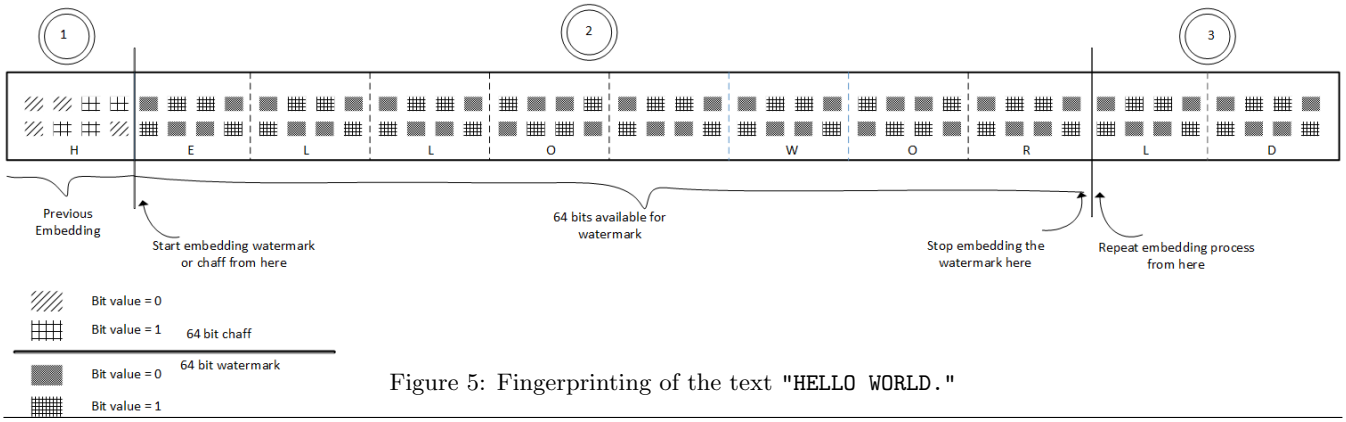


Figure 5: Fingerprinting of the text "HELLO WORLD."

algorithm EmbedWatermark
input version D_i^v **of** document D_i; available watermarking schemes M
output watermarked version $D_i'^v$

Let $[b_1, b_2 \ldots, b_k]$ be the embeddable bits
$S_v^i = \text{SHA256}(\text{PK}(U_i)||\text{ID}(D_i)||v)$
Initialize PRNG **with** S_v^i
for i := 1 **to** k, i+=64 **do**
  if coin flip = embed watermark **then**
    Randomly select an embedding scheme M_i from M
    $[\langle b_i, b_{i+1}, b_{i+2}, \ldots, b_{i+64}\rangle]$:= next random number
    using M_i
  else
    $[b_i, b_{i+1}, b_{i+2}, \ldots, b_{i+64}]$:= random chaff
  end if
end for

Figure 6: Watermark embedding algorithm.

algorithm ExtractWatermark
input watermarked version $D_i'^v$; available watermarking schemes M; threshold δ
output possible traitor U_i

Let UserList[1...n] = {0, 0, 0..., 0}
Let BitErrors[1...n] = {−1, −1, ..., −1}
Let $[b_1, b_2, \ldots, b_m]$ be the embedded bits **in** the document
inverted_bits_count := 0
window_size := 64
for p := 1 **to** m, p+=window_size **do**
  for q := 1 **to** n, q+=1 **do**
    extracted_number := $\langle b_p, b_{p+1}, b_{p+2}, \ldots, b_{p+window_size}\rangle$ using scheme M_q
    if extracted_number ≈ a number **in** user $U_i's$ number sequence **and**
      inverted_bits_count $\leq \frac{\delta}{100} *$ window_size **then**
      UserList[U_i] := UserList[U_i] + 1
      if BitErrors[U_i] = −1 **then**
        BitErrors[U_i] := inverted_bits_count
      else
        BitErrors[U_i] := BitErrors[U_i] + inverted_bits_count
      end if
    end if
  end for
end for
return user **with** maximum value **in** UserList **and** minimum value **in** BitErrors

Figure 7: Watermark extraction algorithm.

exhaustive scan over the document's embedded carrier bits and all generated watermark numbers is sufficient to extract the fingerprint and identify the user who leaked a document.

The extractor tailors the extraction process to the form of the input document. For example, if the input document is a jpeg file and the embedding scheme is synonym replacement, the document needs to be OCR'ed before watermark bits can be extracted.

Figure 7 shows pseudo-code for the extraction algorithm. The algorithm slides a 64-bit window across the bitstream and each 64-bit block is considered as a possible watermark. This sliding window approach makes it possible to identify watermarks even when text has been moved around the document. The extracted watermark bits are compared against the set of watermarks created for each user within the system.

We set a threshold for the allowable number of bit errors introduced by the scanning process. This can be configured based on the quality of the scanned document and the accuracy of the OCR process. A threshold set to 10 means that the current window is considered a match if up to 10% of its bits have been inverted.

Once we have exhausted all possible sequences obtained by sliding the 64 bit window, the user with most number of document hits and least number of errors is chosen and added to the list of probable traitors. We then repeat this process for all the other watermark schemes and compute the list of probable traitors. Finally, we will have a list of probable traitors and we choose the user with minimal number bit errors and maximum number of matches.

Our current algorithm represents an exhaustive search. This is sufficient in most situations: in most organizations the majority of users are trusted, documents are rarely leaked, and hence traitor-tracing is a very infrequent operation. For large organizations where documents are circulated among many users, and leaking is a major problem, a more efficient extraction algorithm is necessary. For example, the search space can be reduced by only considering those documents with text that matches the leaked document. The algorithm is also trivially parallelizable.

4.4 Embedding Schemes

Our current implementation watermarks Open Office text documents using a combination of character color and word spacing. Other schemes can be easily added.

4.4.1 Character Color Embedding

Following [19], we embed one bit of information in each character's color, setting the fifth highest order bit of the Red and Green components, resulting in a carrier with a capacity of 2 bits. In a real world scenario a leaked document would be in the form of a PDF file, a print-out, or a jepg image. Our extractor uses an image processing library (*Leptonica*) and an OCR engine (*Tesseract*) to process these document types. To extract the embedded bitstream we first convert the document to an image, which is OCR'ed, producing a stream of characters and their location in the document. From the character locations we extract the original character color using a standard color quantization algorithm. The embedded watermark bit is extracted and appended to the extracted watermark.

4.4.2 Word Space Embedding

To embed a watermark in word spacing we keep the default spacing for a 0 bit, and increase the space slightly for a 1 bit. One challenge faced while building this scheme was how Open Office handles words that do not fit onto a line. If the document is not right justified the word gets pushed to the next line. This leaves an undesired space at the end of the last line and there is no way to determine whether the bit embedded is a 0 or 1. We overcame this scenario by maintaining a copy of the previously embedded bit and use this copy in the next word space encountered on the new line. The word spaces that follow from here on will be embedded with the next watermark bit available from the bitstream. To extract the watermark we again use the OCR engine to iterate over the word-positions of the document, compute inter-word gaps, and produce a 0 bit if the gap is close in size to the default word spacing, a 1 bit otherwise.

4.5 Evaluation

To evaluate the embedding schemes we embed marks into the *Declaration of Independence.* On an average, 75 unique 64 bit fingerprints were embedded in each page of the four page document. This document was then exported as a PDF file and passed through the extraction component of the watermarking framework. The actual extraction of the embedded watermark took no more than 10 seconds. After this, the traitor tracing process took an additional 6 hours and 42 minutes to complete successfully in an environment with 5 users of the system and 10 documents of similar size. On an average, 35 fingerprints of the user were recognized per page of the document. Execution time may seem excessive (it is the result of the exhaustive search nature of the process), but traitor-tracing is only performed very infrequently, when a leak is suspected. The execution time can easily be improved by restricting the search set to only those document versions that match the contents of the leaked document and by parallelizing the search.

5. CLIENT-SIDE INTEGRITY

Any system for collecting provenance will be susceptible to so-called *Man-At-The-End* (MATE) attacks. The reason is that users, trusted or malicious (see Figure 2), are in control of the system that collects the provenance and are therefore free to reverse engineer and tamper with its code or internal data structures. For example, previous systems that collect provenance at the application-layer (*Sprov* [23]) or the kernel level (PASS [27]) cannot guarantee metadata integrity in the presence of a malicious insider.

In other words, while the system can use cryptography to protect the confidentiality and integrity of the chain of provenance records, it cannot unconditionally protect *its own* integrity. Thus, ultimately, our trust in a provenance chain becomes a combination of our trust in the cryptographic algorithms used to protect it and our trust in the mechanisms used to protect the integrity of the code that implements those algorithms.

Many techniques have been devised to protect software against integrity violations [16]. They typically rely on obfuscating code transformations to slow down reverse engineering efforts and tamperproofing to detect or prevent code modifications. A common technique is to insert a network of *guards* [10,24], functions that check that a hash over the binary code returns the expected value and, if not, take appropriate actions such as repairing or aborting. Unfortunately,

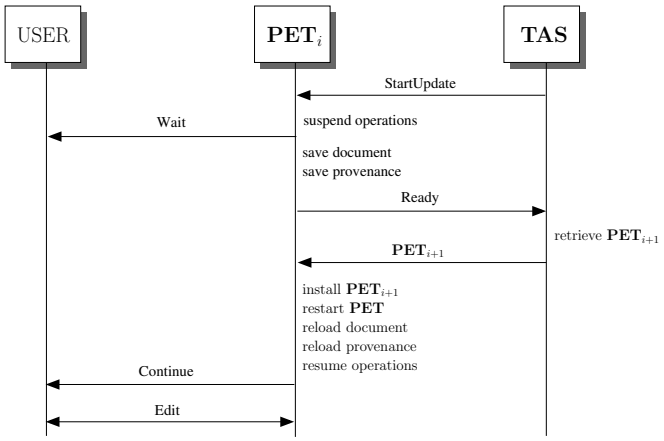


Figure 8: Code update protocol to protect against Man-At-The-End attacks.

no software protection mechanism can provide protection against a determined adversary for an extended period of time. Particularly, if guards are too similar they can be easily found and disabled [5]. Guards can also be disabled using a modification to the operating system [33], an attack which, in turn, can be made ineffectual with a clever use of self-modifying code [22].

5.1 Continuous Renewability and Diversity

It should be noted that, in a distributed system such as Haathi where there exists a trusted entity, *detecting* tampering of untrusted clients is often preferable to, and easier than, *preventing* such tampering.

The *Tigress* system [15] provides tamper detection of client code in a distributed system. The basic idea is to keep the client code in constant flux, frequently replacing it with new code variants, thereby reducing the time available for the adversary to reverse engineer and tamper with the code under their control. The level of security afforded by such as system is a function of the update frequency [9], the diversity of the generated code variants, and their resilience to tampering. Security must be traded off against performance, however, since all obfuscating and tamperproofing transformations introduce a performance penalty.

To protect our provenance system against MATE attacks we are currently in the process of designing a *Tigress*-style code renewability and diversity subsystem to generate a sequence of ever-changing **PET** variants. These are generated by the **TAS** and pushed to the user's site at given intervals.

The code update protocol proceeds as shown in Figure 8. First, the *Software Protection Updater* (a part of the **TAS**, see Figure 3) sends the **PET** a request to begin the update cycle. Next, the **PET** saves the current version of the document along with its provenance, and enters a dormant state where the user can no longer perform any editing operations. Next, the **TAS** retrieves a new variant of the **PET** which, for performance, has been generated beforehand, and sends it to the **PET**. The **PET** restarts, reloads the saved document and provenance, and allows the user to resume editing operations. Below we expand on these steps.

The **TAS** uses a heuristic to determine the update frequency and the amount of software protection to add to each **PET** variant. The heuristic takes into account the perceived threat level of the user, the performance degradation resulting from the software protection itself, and the latency of the code update cycle to find a reasonable security/performance trade-off.

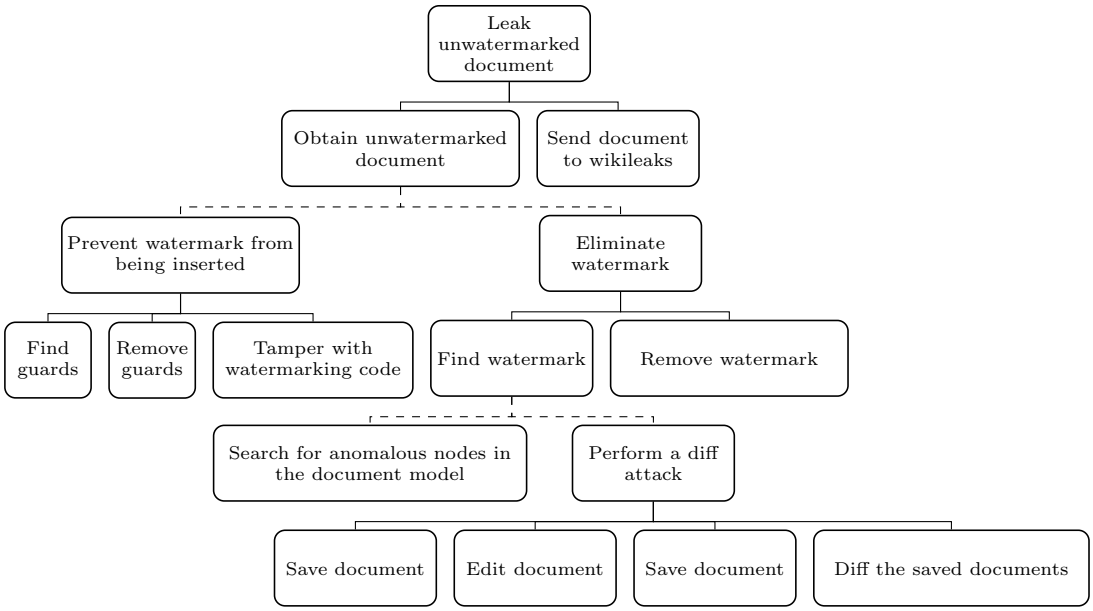
Simply pushing a new variant of the **PET** to the user is not enough, as a malicious user could simply ignore the update in favor of a previously reverse engineered and tampered variant. As in the Tigress system, the generation step therefore also diversifies the application-level communication protocol the **PET** uses to communicate with the **TAS** and the **USS**. A **PET** that chooses to ignore a code update will, eventually, use an outdated protocol which will be detected by the **TAS** or the **USS**. On detection of a misbehaving **PET** the **TAS** can take countermeasures, such as shutting down communication or adding a provenance record indicating reduced trust in the provenance.

6. SECURITY ANALYSIS

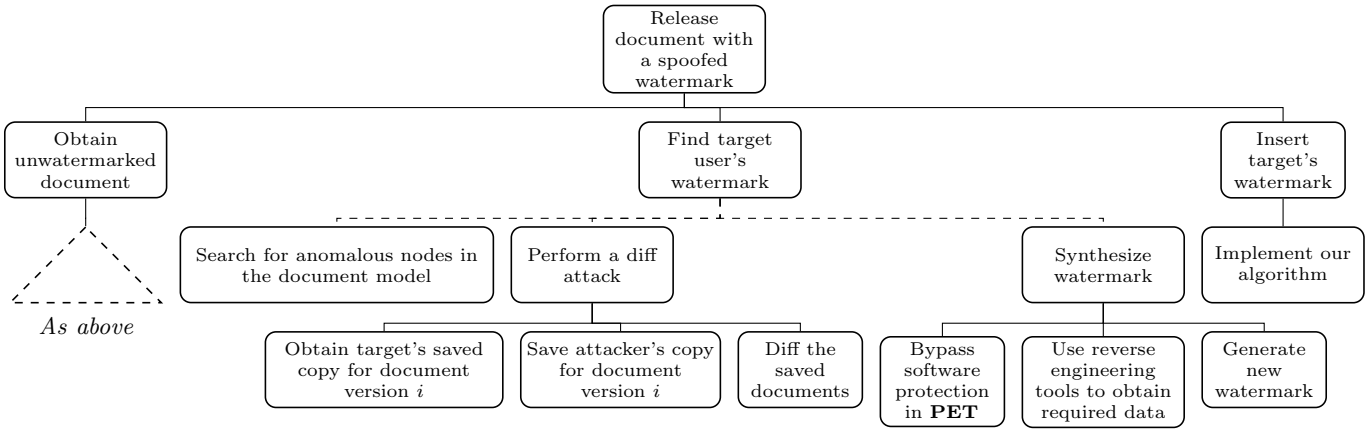
In this section we analyze the possible attacks against Haathi that may result in compromising the traitor detection techniques. In such a scenario, the ultimate goal of an attacker is to leak a document by destroying or tampering with the unique fingerprint embedded within the document. This can be achieved by either eliminating the watermark completely from the document or by spoofing another user's watermark. We examine both possibilities.

6.1 Leaking an unwatermarked document

Figure 9(a) shows an attack tree where the adversary's goal is to leak an de-watermarked document. There are two basic approaches: (1) prevent the document from being watermarked in the first place or (2) eliminate the watermark from a marked document. To prevent Haathi from watermarking the document an adversary must first find and remove the guards without this being detected (Section 5), and then tamper with the watermarking code to ensure that the marks are not inserted. To eliminate the watermarks requires first locating the marks within the document. This can be done in two ways: (1) by searching through the document's object model (the tree data structure that holds the internal representation of the document) for nodes that appear anomalous and removing them or (2) by obtaining



(a) Leaking an unwatermarked document



(b) Spoofing the fingerprint of another user.

Figure 9: Attack trees. **And**-edges are solid, **Or**-edges dashed.

two versions of the same document, taking their binary difference to figure out the location of watermark bits, and eliminating the watermark from all such locations.

To reduce the time available to the attacker to manipulate the **PET** code, our design ensures that the code is in constant flux (Section 5). This also makes it difficult to manipulate the document and its metadata while it is stored internally in the **PET**. Manipulating the data while in transit or stored on the **USS** is not possible since it is kept encrypted.

The document is continuously watermarked with either 64-bit fingerprints or with random chaff, fingerprint numbers are never reused, and the whole document is re-watermarked after every save operation. Taken together, this renders differential attacks impotent.

6.2 Leaking a spoofed document

Figure 9(b) shows a second attack tree where the malicious user's goal is to leak a document that has been spoofed with the fingerprint of a target user. This requires first obtaining an unwatermarked document (using the same methods described above), then finding or generating a set of the target user's fingerprints, and finally embedding these marks in the document using the same set of schemes as chosen by the `EmbedWatermark` algorithm in Figure 6.

Obtaining the target user's fingerprint could conceivably be done by taking the binary difference of two copies of the same document, belonging to attacker and the target user, respectively. However, since marks are randomly scattered over the document, and every embeddable bit could be either chaff or belong to a fingerprint, this is not feasible.

An alternative attack is to bypass the software protection techniques in Section 5, looking "inside" the executing **PET** using, for example, a debugger or other reverse engineering

Table 1: Performance measurements.
(a) Automatic entering of text at the rate of 240 words per minute

Input	**Size** (Sz) (bytes)	N_{prov}	**Time** (T) (sec)	**Space** (S) (bytes)	Sz/N_{prov} (bytes)	T/N_{prov} (msec)	S/N_{prov} (bytes)
Decl. Indep.	34,162	8,659	253	6,935,859	3.94	29.21	801
Hamlet	194,357	194,359	6,044	155,681,559	0.99	31.09	801

(b) Cautious vs. obsessive saver.

User Type	N_{prov}	**Time** (T) (sec)	N_{saves}	**Total Save time** (sec)	**Avg. Save time** (sec)
Lazy saver	8659	253	1	12	12
Cautious saver	8690	334	33	86	2.60
Obsessive saver	8692	338	36	86	2.38

tools, and thereby obtaining document and user ID's and the chosen marking schemes. From these a set of new valid fingerprints could be synthesized. To prevent such attacks frequent updates to the **PET** code, as described in Section 5, is therefore necessary.

7. USABILITY ANALYSIS

Haathi collects extremely fine-grained provenance down to editing events such as adding and deleting individual characters and exposure events such as export and printing. For each provenance event a provenance record is created which, similar to *Sprov* [23], is encrypted, signed, and checksummed. Because of the fine-grained nature of our provenance collection provenance can grow rapidly, and our **USS** is designed to run and store documents and provenance on public cloud servers, in our case Amazon's EC2.

Haathi is designed to be end-to-end secure but usability is no less important than security: we must ask whether the overheads of encryption and remote storage of provenance chains will affect the interactive nature of a document editing system. To test this, we implemented an Open Office plugin that builds a list of provenance records by listening to user key strokes and events such as Export, Copy, Paste, and Save. The encrypted records are sent over to the **USS** on a Save operation. We automated the entering of text and tuned it to enter data at the rate of 240 words per minute, which is faster than the most skilled typist can sustain. The system was tested using two inputs: the Declaration of Independence and Shakespeare's Hamlet.

Our results are given Table 1(a). Here, **Size** is the input document's size; N_{prov} is the number of provenance events recorded; **Time** and **Space** give the overheads for provenance collection. The last three columns of the table give per-event statistics for provenance data. It can be seen that each provenance event incurs a runtime overhead of between 29 and 32 msec, and that each provenance record takes 801 bytes of space. Interestingly, the per-event overheads decrease as the input file size and the number of provenance events increases, suggesting some amortization of overheads. These statistics indicate that response time is well within the needs of interactive use.

In the experiment above we considered a *lazy saver* who saves the document only after completion. We also considered an *obsessive saver* who saves after every sentence and a *cautious saver* who saves after every paragraph. Our results for the Declaration of Independence are given in Table 1(b). Here N_{prov} is the number of provenance events recorded, N_{saves} is the number of saves recorded, and **Time** gives the overheads for provenance collection. We see that the time taken to enter the document does not change drastically based on the user type. Irrespective of the type of user the average time taken to save the document is 2.40–2.60 seconds which is adequate for interactive use.

8. CONCLUSIONS

We present a practical design of a document processing system that collects provenance information while maintaining end-to-end security and interactive usability. Our design uses continuously updatable software protection techniques to protect against Man-At-The-End attacks, and a generic, continuous marking, collusion-free text fingerprinting algorithm to trace documents that, in spite of the software protection, escapes the system.

A prototype of our design (not including the software protection part) has been implemented as an extension to the OpenOffice document management system. To allow for extremely fine-grained provenance collection, which may result in rapid growth of provenance chains, documents and their provenance are stored encrypted on public cloud servers. Analysis shows that in spite of the extra work, interactivity does not suffer.

Acknowledgments This work was supported by the US-Israel Binational Science Foundation under grant 2008362.

9. REFERENCES

[1] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance collection support in the kepler scientific workflow system. In Luc Moreau and Ian T. Foster, editors, *IPAW*, volume 4145 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2006.

[2] M. Atallah, V. Raskin, C. Hempelmann, M. Karahan, R. Sion, and K. Triezenberg. Natural language watermarking and tamperproofing. In *Proc. 5th International Information Hiding Workshop*, 2002.

[3] Mikhail J. Atallah, Victor Raskin, Michael Crogan, Christian Hempelmann, Florian Kerschbaum, Dina Mohamed, and Sanket Naik. Natural language watermarking: Design, analysis, and a proof-of-concept implementation. In *IHW '01: Proceedings of the 4th International Workshop on Information Hiding*, pages 185–199, London, UK, 2001. Springer-Verlag.

[4] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs (extended abstract). In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, 2001. LNCS 2139.

[5] Philippe Biondi and Fabrice Desclaux. Silver needle in the skype. In *Black Hat Europe*, Feb-Mar 2006. `www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf`.

[6] Paulo Vinicius Koerich Borges and Joceli Mayer. Text luminance modulation for hardcopy watermarking. *Signal Process.*, 87(7):1754–1771, July 2007.

[7] J.T. Brassil, S. Low, and N.F. Maxemchuk. Copyright protection for the electronic distribution of text documents. *Proceedings of the IEEE*, 87(7):1181–1196, 1999.

[8] A. Castiglione, B. DâĂŹAlessio, AD Santis, and F. Palmieri. Hiding information into ooxml documents: New steganographic perspectives. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 2(4):59–83, 2011.

[9] Mariano Ceccato and Paolo Tonella. Codebender: Remote software protection using orthogonal replacement. *IEEE Software*, 28(2):28–34, 2011.

[10] Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, Philadelphia, PA, USA, November 2001. Springer Verlag, LNCS 2320.

[11] Q. Chen, Y. Zhang, L. Zhou, X. Ding, and Z. Fu. Word text watermarking for ip protection and tamper localization. In *Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC), 2011 2nd International Conference on*, pages 3595–3598. IEEE, 2011.

[12] Benny Chor, Amos Fiat, and Moni Naor. Tracing traitors. In *Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '94, pages 257–270. Springer-Verlag, 1994.

[13] C. Collberg, E. Carter, S. Debray, J. Kececioglu, A. Huntwork, C. Linn, and M. Stepp. Dynamic path-based software watermarking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 04)*, 2004.

[14] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *ACSAC*, 2012.

[15] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Remote tamper detection. 2012.

[16] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Software Security Series. Addison-Wesley, July 2009.

[17] I. J. Cox, M. L. Miller, and J. A. Bloom. *Digital Watermarking: Principles and Practice*. Morgan Kaufmann, 2002.

[18] Susan Davidson, Zhuowei Bao, and Sudeepa Roy. Hiding data and structure in workflow provenance. In *Proceedings of the 7th international conference on Databases in Networked Information Systems*, DNIS'11, pages 41–48, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] M. Du and Q. Zhao. Text watermarking algorithm based on human visual redundancy. *AISS: Advances in Information Sciences and Service Sciences*, 3(5):229–235, 2011.

[20] Funda Ergun, Joe Kilian, and Ravi Kumar. A note on the limits of collusion-resistant watermarks. In *Advances in Cryptology, Eurocrypt '99*, volume 1592, pages 140–149, 1999.

[21] Juliana Freire, Cláudio T. Silva, and Emanuele Santos Steven P. Callahan, Carlos E. Scheidegger, and Huy T Vo. Managing rapidly-evolving scientific workflows. In *Proceedings of the 2006 international conference on Provenance and Annotation of Data*, IPAW'06, pages 10–18, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] Jonathon Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, pages 18–27, Tucson, AZ, USA, December 2005. Applied Computer Associates, IEEE.

[23] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[24] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001*, Philadelphia, PA, USA, November 2001. Springer Verlag, LNCS 2320.

[25] H.Y. Kim and J. Mayer. Data hiding for binary documents robust to print-scan, photocopy and geometric distortions. In *Computer Graphics and Image Processing, 2007. SIBGRAPI 2007. XX Brazilian Symposium on*, pages 105–112. IEEE, 2007.

[26] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E. McGrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In Juliana Freire, David Koop, and Luc Moreau, editors, *IPAW*, volume

5272 of *Lecture Notes in Computer Science*, pages 323–326. Springer, 2008.

[27] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track*, pages 43–56. USENIX, 2006.

[28] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Tim Carver, Matthew R. Pocock, and Anil Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20, 2004.

[29] Sudha Ram and Jun Liu. A new perspective on semantics of data provenance. In *Proceedings of the First International Workshop on the role of Semantic Web in Provenance Management (SWPM 2009)*, 2009.

[30] A.L. Varna, S. Rane, and A. Vetro. Data hiding in hard-copy text documents robust to print, scan and photocopy operations. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 1397–1400. IEEE, 2009.

[31] R. Villań, S. Voloshynovskiy, O. Koval, F. Deguillaume, and T. Pun. Tamper-proofing of electronic and printed text documents via robust hashing and data-hiding. pages 65051T–65051T–12, 2007.

[32] R. Villàn, S. Voloshynovskiy, O. Koval, J. Vila, E. Topak, F. Deguillaume, Y. Rytsar, and T. Pun. Text data-hiding for digital and printed documents: Theoretical and practical considerations. In *In Proceedings of SPIE-IS&T Electronic Imaging 2006, Security, Steganography, and Watermarking of Multimedia Contents VIII*, 2006.

[33] Glen Wurster, P.C. van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *2005 IEEE Symposium on Security and Privacy*, pages 127–138, 2005.