`Watermill`: an optimized fingerprinting system for highly constrained data

Camelia Constantin
OASISLIP 6
Université Pierre et Marie Curie
8 rue du Capitaine Scott
75015 Paris, France
camelia.constantin.(at).lip6.fr

David Gross-Amblard *
Laboratoire Cedric
Spécialité Informatique
CC 432
Conservatoire national des arts & métiers
292 rue Saint Martin
75141 PARIS Cedex 3, France
dgram.(at).cnam.fr

Meryem Guerrouani
Laboratoire Cedric
Spécialité Informatique
CC 432
Conservatoire national des arts & métiers
292 rue Saint Martin
75141 PARIS Cedex 3, France

Abstract

Relational databases watermarking aims at protecting the intellectual or industrial property of a dataset, by applying secret and slight alterations on it. When critical usability constraints of this dataset must be preserved, finding such alterations (watermarks) is a difficult computational task, which is not optimized by the current watermarking systems. This is a critical limitation when considering *fingerprinting* applications, where several distinct watermarked databases have to be obtained.

An important property of the watermark is to be resilient to attacks that try to erase it. Among these attacks, one of the most severe is the *collusion attack*, that locates the watermark by comparing several distinct watermarked versions of a database. Such an attack has not been taken into account by the existing databases watermarking methods, when usability constraints have to be preserved.

*Work supported in part by the ACI Sécurité & Informatique grant - TADORNE project (2004)

In this paper, we present an efficient algorithm for collusion-secure fingerprinting that preserves usability constraints. We identify a class of constraints, namely *weight-independent constraints*, that can be translated into an integer linear program. Solutions of this program are acceptable watermarks. This representation is computed once and for all and has the following advantages. First, we can rely on state-of-the-art optimizers to reduce the search space and quickly find a good watermark. Second, for a carefully chosen class of watermarks, producing every new watermark is immediate, compared with the complete recomputation needed by existing watermarking methods. Finally, our algorithm addresses the problem of collusion attacks, by making possible the use of collusion-secure codes while respecting usability constraints.

The formalism of this article permits to handle both relational databases and XML documents. The effectiveness of our techniques has been established on our open platform `Watermill`, that integrates fingerprinting management capabilities to standard relational or XML-native databases systems.

1 Introduction

Databases watermarking The growth of the Internet and the World-Wide-Web has been followed by a drastic increase of digital data exchange. Users are searching for valuable data, whose elaboration costs time and money. In this setting, data owners are exposed to malicious users who disseminate unauthorized copies of a copyrighted work.

Watermarking aims at strengthening ownership proofs by hiding copyright information (called *watermark*) into the purchased document. The watermark should be invisible, i.e. should not impact the document usefulness. It must also be robust against attacks performed by malicious users trying to erase it.

Watermarking techniques are widely studied in the area of multimedia documents (see [6, 10] for a survey), and several recent efforts have lead to complete solutions for the watermarking of relational databases [4, 15]. In all these solutions, watermarking is done by altering the data. This is obviously a limitation for a legitimate purchaser, but it is well known that these alterations are necessary to achieve watermark robustness. In multimedia watermarking, for example in images watermarking, alterations are limited so that the image quality is preserved. This quality can be expressed by signal processing characteristics, like luminance. For databases watermarking, even small alterations on data may ruin the result of specific queries (e.g. join queries) that are important for the database purchaser. Hence specific tools for defining databases usefulness according to the purchaser's need are required.

Usability constraints Usability constraints (also known as *semantic integrity constraints* [17, 7]) are used to specify data properties that are crucial from the database purchaser's point-of-view. They are designed by the database owner, based on the purchaser's application specification during a negotiation phase with the purchaser. For example, a purchaser may require that joins between some attributes should be preserved, or that some attributes will be used for querying data on the purchaser's side. Part of these constraints may be public, so that the purchaser knows the limit of its data. Part of them may be kept secret to enforce the watermark invisibility. An important point is that the database owner may refuse to sell its database to a purchaser who asks for usability constraints leading to non-watermarkable data (e.g. a purchaser requiring no alteration at all). However, it is certain that a purchaser will prefer data with controled alterations rather than no data at all.

Finding watermarks: the greedy method The main problem is then to find robust watermarks that respect the given usability constraints. A natural method is the *greedy search* [15]. For a given secret binary message to encode, its first bit is embedded by distorting a small set of tuples. Then, usability constraints are checked on all the dataset, involving the computation of several queries. If usability constraints are preserved, the following bit is considered. If usability constraints are not respected, changes are discarded on the current set of tuples, and another set is considered.

Watermarking speed and fingerprinting Experimental evidence shows that the greedy method may find acceptable watermarks (distortions) in one pass, but sometimes, no valid watermark is found, and backtrack is necessary. Hence finding acceptable watermarks is difficult and computationnaly intensive, since all usability constraints have to be tested for each hidden bit.

A natural extension of watermarking is *fingerprinting*, where a different watermarked data is distributed to each member of a group of purchasers [18] sharing the same usability constraints. A classic application of this technique is traitor tracing, i.e. proving which purchaser is the source of an illegal diffusion. This extension is challenging, since now several acceptable watermarks are to be computed.

For this application, the computational effort of the greedy method is tremendous. Each constraint is checked on each bit insertion, and for T watermarked versions, this expensive operation must be iterated T times.

One may argue that since watermarking is done once and for all in the life-cycle of a document, speed is not an issue. On the contrary, we think that speeding up the watermarking process may become a real requirement. Indeed, several onerous datasets like meteorological measurements [3] have a huge commercial or scientific value, but during a finite time window. For example, weather forecasts for the next week are not valuable anymore after the current week. Thus, if watermarking for ten purchasers needs five days of computation to proceed, its applicability is reduced. Also, due to this limited time window, caching techniques can not be used. So, for critical applications, watermarking time complexity is an important issue and should be lowered.

Our contribution In this paper, we propose an optimized watermarking and fingerprinting method. To do so, we provide a declarative language for usability constraints definition. Then we locate specific patterns in these usability constraints where optimization can be performed. These patterns include aggregate and join computations, which are central in the design of usability constraints used in real-world datasets [13]. It should be noted that this approach, in the spirit of databases query optimization, was not addressed by previous works. Note also that the optimization techniques we use for watermarking are different from those used in standard query optimization.

This technique leads to the following new capabilities (that will be defined and discussed in the following): (1) speeding-up the watermarking/fingerprinting process and (2) resisting attacks from a collusion of purchasers *while respecting usability constraints.*

Speeding-up watermarking and fingerprinting We decompose the watermarking process in two phases. *(i)* In the first, *compilation phase*, we analyze the usability constraints and the database given by the owner, looking for specific constraint patterns, namely *weight-independent constraints.* Matching constraints will be translated into an integer linear program (ILP) whose solutions are potentially good watermarks. Constraints that do not match these patterns will be treated by the

greedy method. *(ii)* The second phase is the proper *watermarking phase*. It consists in finding one or several solutions to the previous linear program and modifying the database accordingly.

In our framework, once the compilation phase is over, producing a watermarked version benefits from state-of-the-art techniques from integer linear programming. Furthermore, if we restrict our attention to a carefully chosen class of watermarks, finding a large number of watermarks for fingerprinting is immediate. Our experiments show (section 7) that the computation time can be greatly lowered with this method.

Collusion-secure fingerprinting When distributing several fingerprinted versions of a database, the owner is exposed to collusion attacks. In this setting, several malicious purchasers may collude to compare their watermarked versions. By locating positions where their documents differ, they can discover where watermark bits have been inserted. By modifying documents on these positions, they may obtain a new version without a readable watermark, hence evading detection. The design of efficient collusion-secure fingerprinting codes is a long-standing effort [5, 9, 16]. In order to be collusion-secure, watermark messages must be carefully chosen from a precise *codebook*, and inserted in the *same* positions in all the distributed versions.

We show that for the aforementioned weight-independent constraints, a family of good watermarks can be quickly found, so that bit encoding is performed always on the same positions. Hence using a collusion-secure code is possible, while preserving usability constraints.

Outline of the paper Section 2 recalls basic concerns about watermarking, and introduces XML watermarking as a natural extension. Our declarative language for usability constraints is defined in section 3. We then study the optimization of watermark discovery for specific kinds of usability constraints: by using pure integer linear programming in section 4, and by focusing on a specific family of solutions in section 5. Collusion-secure fingerprinting is addressed in section 6. Section 7 presents experiments on our open prototype `Watermill` [1], that integrates fingerprinting management capabilities to standard relational or XML-native databases systems.

Related work Several recent works consider relational databases watermarking [4, 15, 8, 12, 11]. Agrawal and Kiernan's method [4] hides information in the least significant bits (LSB) of numerical attributes. The database owner can control the alteration on attributes by setting the number of LSB that can be modified. Although a small overall distortion on the mean of the watermarked attributes is observed, more general usability constraints are not considered. Their technique was extended [12] to collusion-resilient fingerprinting by using collusion-secure codebooks [5]. But again, usability constraints are not handled.

Sion, Atallah and Prabhakar [15] have introduced the greedy method for watermarking with usability constraints. They handle potentially any constraint type by repeatedly calling external checking programs (*usability plugins*). This very general method is not optimized in the sense used in the introduction, as the syntactical form of usability constraints is not explored. Their method applies also for fingerprinting, but the computational effort is tremendous. Combining their method with collusion-secure codebooks is also possible but with limitations: there is absolutely no guarantee that the same watermark positions will be found for all watermark messages in the codebook since their method is greedy.

Weight-independent *sum* constraints were considered by one of the authors of the present paper from the theoretical point of view [8]. This specific query pattern is studied in order to obtain

```
<mills year=2003>
  <windmill> <place>Bretagne</place>
             <x>30</x><y>53</y>
             <prod>125</prod>
  </windmill>
  <windmill> <place>Vendee</place>
             <x>62</x><y>56</y>
             <prod>223</prod>
             <height>90</height>
  </windmill>
  <mill>     <x>55</x><y>22</y>
             <prod>443</prod>
             <height>5</height>
  </mill>
  <mill>     <x>22</x><y>51</y>
             <prod>53</prod>
             <height>2</height>
  </mill>
  <geothermic><place>Dijon</place>
              <prod>33</prod>
              <prod>66</prod>
  </geothermic>
</mills>
```

Figure 1: mills.xml

a lower bound on the number of distinct acceptable watermarks that one is likely to discover. The algorithmic counterpart of this previous paper is less suited for practical applications. The present paper considers better algorithms that behave correctly with large datasets. Moreover, the algorithms of this previous work were not *blind*, while our new algorithms are. Finally, linearizable constraints and collusion attacks were also not considered.

2 Databases and XML watermarking

2.1 Example

Semi-structured documents The following example is used throughout the paper. A data owner has spent time and effort to build the following accurate document `mills.xml` (Figure 1), that represents different kinds of powerplants on a map. A powerplant can be a windmill, a mill or a geothermic installation. Other elements are the `(x,y)` GPS positions on this map, the `place` where they are located, energy `prod`uction measures carried out on these units during year 2003, and the `height` of the installation.

This document respects the schema and key constraints of Figure 2. Consequently, any `x`, `y`, `prod` or `height` element for a windmill can be uniquely determined by the value of the `place` element. This point will be central in the sequel.

Schema for `mills.xml`:

```
mills -> (windmill|mill|geothermic)*
windmill -> place x y prod (height)?
mill -> x y prod height
geothermic -> place prod*
```

Key constraints for `mills.xml`:

```
<xs:key name="key1">
   <xs:selector xpath="/mills/windmill"/>
   <xs:field xpath="/place"/>
</xs:key>
<xs:key name="key2">
   <xs:selector xpath="/mills/mill"/>
   <xs:field xpath="/x"/>
   <xs:field xpath="/y"/>
</xs:key>
```

Figure 2: Schema and key constraints for mills.xml

Watermarking A *data server* (e.g. a Web service) may be interested in such data, to provide information about powerplants on a Web site. This server answers queries to *final users*. A malicious server may copy the document and try to sell it on his own.

To prevent this, the aim of watermarking is to hide information in the original document in order to prove ownership once a suspect document is discovered. The data owner will insert a *watermark* into the original document, i.e. will slightly modify several element values. For the sake of simplicity, we restrict ourselves to real number values, but other types can be considered. As an example, documents `mills2.xml` and `mills3.xml` of Figure 3 are examples of watermarked copies of `mills.xml`. They differ on several positions, e.g. `prod` and `height` elements of windmill `Vendee`.

When a suspect copy is discovered, the owner will extract the watermark from the suspicious document, and compare it to the inserted one. If these two watermarks are similar (in a way defined later), he can claim ownership of the document. We call the *marker* the algorithm for watermark insertion, and *detector* the algorithm for watermark detection.

Adversarial model In a naive setting, the stolen document is kept identical with the purchased one. In this case, the two watermarks are identical, and the ownership proof is strong. In a more realistic, *adversarial setting*, the purchaser may alter the stolen document (up to a realistic extent) in order to erase the watermark.

Classic types of attack are:

- data alteration: In the *random data alteration attack*, a subset of data is randomly distorted. The amplitude of this alteration is limited so that the dataset is still valuable. In the *rounding attack*, least significant bits of the data are erased;
- *data loss*: a subset of data is suppressed;
- *mix-and-match*: new data from another source is added to the dataset.

mills2.xml

```
<mills year=2003>
  <windmill>
    <place>Bretagne</place>
    <x>35</x><y>53</y>
    <prod>127</prod>
  </windmill>
  <windmill>
    <place>Vendee</place>
    <x>62</x><y>56</y>
    <prod>203</prod>
    <height>91</height>
  </windmill>
  <mill>
     <x>55</x><y>22</y>
     <prod>463</prod>
     <height>5</height>
  </mill>
  <mill>
     <x>22</x><y>51</y>
     <prod>53</prod>
     <height>2</height>
  </mill>
  <geothermic>
     <place>Dijon</place>
     <prod>33</prod>
     <prod>66</prod>
  </geothermic>
</mills>
```

mills3.xml

```
<mills year=2003>
  <windmill>
     <place>Bretagne</place>
     <x>40</x><y>53</y>
     <prod>125</prod>
  </windmill>
  <windmill>
     <place>Vendee</place>
     <x>62</x><y>56</y>
     <prod>203</prod>
     <height>90</height>
  </windmill>
  <mill>
     <x>55</x><y>22</y>
     <prod>423</prod>
     <height>5</height>
  </mill>
  <mill>
     <x>22</x><y>51</y>
     <prod>63</prod>
     <height>2</height>
  </mill>
  <geothermic>
     <place>Dijon</place>
     <prod>33</prod>
     <prod>66</prod>
  </geothermic>
</mills>
```

Figure 3: Two watermarked instances

Hence a good watermarking system should be robust against such alterations. As in [4, 15, 8], *we suppose that any attack on the document keeps keys unchanged.* This assumption is natural when keys have a public meaning. For example, the `windmill` element with `<place>Bretagne</place>` can not be changed to `<place>Paris</place>` without misleading all users of a stolen dataset.

Watermarking identified values The main hypothesis, used also in [8, 15, 4] in the relational setting, is that modified values (i.e. watermark positions) must be in the scope of a primary key. This way, modified values are clearly identified, and this helps watermark recovery in the adversarial setting (this is one of the main differences with classic multimedia watermarking, where keys do not exist.)

Let d be an XML document. A leaf element e in d is said to be *identified* if this precise element e can be pinpointed in the entire document using a *key constraint* (ks, kp, ke) and a *key value* kv:

- (ks, kp) are respectively key selector and key path in a key constraint;
- kv is a key value;
- ke is a path relative to element $ks[kp = kv]$ leading to element e.

In our example, the `height` of the windmill of `Vendee` is clearly identified, with ks = `/mills/windmill`, kp = `/place`, kv = `Vendee` and ke = `height`. Its value is 90 in the original document `mills.xml`, and 91 in `mills2.xml`.

We denote by $\mathcal{I}(d)$ the set of identified values of the document d. In our example, $\mathcal{I}$(`mills.xml`) contains all `prod` elements for key values in $\{Bretagne, Vendee, (55, 22), (22, 51)\}$ (recall that for a `mill` element, keys are `x` and `y` elements). The two elements `prod` from the `geothermic` node are not listed, since they is no way to differentiate them using keys. The set $\mathcal{I}$(`mills.xml`) contains also `height` elements.

We sometimes use key values to denote an identified element: for example $\texttt{prod}_{\texttt{Bretagne}}$ is the unique element `prod` identified by the key value `Bretagne`. For $N = |\mathcal{I}(d)|$, we can also consider the set of identified elements as a large vector $\vec{v}(d)$, where for all $i \in \{1, \ldots, N\}$, $\vec{v}(d)[i]$ is the value of the identified element i.

Watermarking method (starting point) We recall here the Agrawal and Kiernan's method [4] for the watermarking of relational databases. We expose this method in the XML setting (which is immediate), and we will use it as a reference for the subsequent algorithms. Note also that the different method from Sion et al. [15] can also be used, but we will not consider it due to lack of space.

The method relies on a pseudo-random generator $\mathcal{S}$ whose production are difficult to predict if one does not know the secret *seed*. Several parameters are used: the secret key $\mathcal{K}$, the ratio γ of watermarked elements, the maximum number ξ of least significant bits (LSB) one can distort, and the maximum probability α of detection errors[1].

The algorithm respects the following steps: for each identified elements i, the random number generator is seeded with $\mathcal{K}$ concatenated with i. If the first number produced then by $\mathcal{S}$ is $0 \mod \gamma$,

[1] The number ν of relational attributes available is also used, but this notion disappears here since we consider XML documents as a flat vector of identified values. Hence $\nu = 1$.

then the value is considered for watermarking. In this value, a bit position is chosen according to the next integer from $\mathcal{S}$, computed modulo ξ. This bit position will be replaced in the value by a mark bit. Finally, the value of this mark bit is given by the parity of the third production of $\mathcal{S}$.

The detection algorithm proceeds identically by locating bit positions in a suspect documents. The found bit mark is then compared with the intended one, and the number of correct matches is recorded. If the match ratio exceeds a given threshold (which is a function of α), the document is declared suspect.

This method exhibits several important properties: robustness, accuracy, incremental updatability, public system and blindness (see [4] for a complete discussion). Among them, blindness means that the original dataset is not needed for detection: only the watermarking parameters $(\mathcal{K}, \gamma, \xi, \alpha)$ and the suspect dataset are required. This is critical for very large documents watermarking, since their backup, copy or transmission to a trusted authority may not be easy.

Usability constraints Purchasers, e.g. a group of Web servers, buying the document `mills.xml`, may answer queries on the document for final users, as "total production during year 2003 for windmills and mills powerplants". We refer to this particular query as query ψ_1, expressed in XQuery:

$$\psi_1 \equiv \texttt{for \$a in /mills/(windmill|mill)} \\ \texttt{return sum(\$a/prod)}.$$

Since data will be modified by watermarking, the result of ψ_1 may change on watermarked documents, and final users of the Web sites might be impacted. Hence, before the owner watermarks the document, owner and servers must agree on *usability constraints* to respect, in order to control the impact on final users. As in [15, 8], we distinguish between the set of *local usability constraints* $\mathcal{L}$, that concern basic elements values, and the set of *global usability constraints* $\mathcal{G}$, that apply on a whole subdocument.

In our example, for accuracy purposes, we may apply the following local constraints $\mathcal{L} = \{C_1, C_2, C_3\}$:

$$C_1 \equiv \text{map positions should not be modified by more than 10 units;}$$

$$C_2 \equiv \text{height of windmill should not be altered by more than 1 meter;}$$

$$C_3 \equiv \text{a production measure is bound to a 20 KW error.}$$

An element e is a *modifiable value* of d if e is in the scope of at least one local constraint in $\mathcal{L}$ (a formal definition is given in the next section). In this paper, watermark insertion is done only on elements values that are **both identified and modifiable**. We denote by $\mathcal{M}(d)$ the set of these elements. By definition, $\mathcal{M}(d) \subseteq \mathcal{I}(d)$.

We now enrich our example with a set of global constraints $\mathcal{G} = \{C_4, C_5, C_6, C_7, C_8, C_9\}$. Constraint C_4 controls the variation of query ψ_1:

$$C_4 \equiv \text{result of query } \psi_1 \text{ should not vary more than 10 KW.}$$

Constraint C_5 expresses that, for legal reasons, productions under 60 KW should be still under this limit *after* watermarking.

$$C_5 \equiv \begin{array}{l}\text{productions lower than 60 KW should re-}\\ \text{main under this limit.}\end{array}$$

Constraint C_6 imposes that the distance between the Bretagne powerplant and a power collector located in position $(10, 10)$ on the map should not be distorted by more than 5 units.

$$C_6 \equiv \begin{array}{l}\text{distance between Bretagne and (10,10)}\\ \text{should not vary more that 5 units.}\end{array}$$

Constraint C_7 states that production units with equal heights should still have equal heights after watermarking (but not necessarily with the same value).

$$C_7 \equiv \begin{array}{l}\text{the set of powerplants that join on height}\\ \text{should remain the same.}\end{array}$$

This kind of constraints is very useful to preserve foreign key relationships between parts of the document.

Calls to an external checking program on the owner's side can be used.

$$C_8 \equiv \begin{array}{l}\text{program } \texttt{qualityChecker} \text{ should accept}\\ \text{the watermarked dataset.}\end{array}$$

Finally, suppose that heights are expressed in *feet* for a windmill, and in *meter* for a mill. Constraint C_9 expressed that, for administrative reasons, the sum of two specific powerplants'height should not exceed a given limit in meter.

$$C_9 \equiv \begin{array}{l}\text{Height of } \texttt{Vendee} \text{ and mill located at}\\ \texttt{(55,22)} \text{ should not exceed 30 meters.}\end{array}$$

A watermark is a *good* watermark if it preserves both local and global constraints defined in $\mathcal{L}$ and $\mathcal{G}$. Finding a good watermark may be a difficult task, as we will see in the sequel.

In Figure 3, you may observe that documents `mills2` and `mills3` respect local constraints C_1,C_2 and C_3. Document `mills2` respects global constraints C_4 and C_5, but not document `mills3` (because the overall production variation is bigger than 10, and because a production jumps over 60.)

Relational setting While this paper is focused on XML documents, its techniques apply also in the relational setting. Indeed, all watermarking operations (insertion, detection) are done on the vector structure $\vec{v}(d)$. If primary keys identifying attributes are available in the relational database, a corresponding vector structure can be constructed (if primary keys are not available, pseudo-keys can be constructed [4]).

3 Declarative watermarking constraints

Example In order to ease the owner's work, we provide a simple declarative formalism to express usability constraints. Before giving a formal semantic, we begin by translating constraints from the previous example:

```
local 10 on //x, //y,                # C1
local 1 on //windmill/height,        # C2
local 20 on //prod,                  # C3
global 10 on (                       # C4
   for $a in /mills/(windmill|mill)
   return sum($a/prod)
),
invariant ($a in /mills/(mill|windmill))
where $a/prod < 60,                  # C5
global 5 on (                        # C6
 for $a in /mills/windmill
 where $a/place=Bretagne
 return sqrt(sqr($a/x-10)+sqr($a/y-10))
)
invariant($a in /mills/mill/height,    # C7
          $b in /mills/mill/height)
where  $a/height = $b/height,
check "qualityChecker"                 # C8
linear(                                # C9
  $a=/mills/windmill[place=Vendee]/height,
  $b=/mills/mill[x=55][y=22]/height,
      0.304 * $a + $b < 30)
)
```

Semantic We now give the formal semantic of these constraints. In the sequel, $\varphi(d)$ denotes the resultset of query φ on document d. Given an XPath query φ, its *identified part* $\mathcal{I}(\varphi, d)$ is the subset of elements in $\varphi(d)$ that is contained in $\mathcal{I}(d)$. This property can be easily and statically checked.

- A `local` constraint has the following form:

  $$\Gamma \equiv \texttt{local}\ p\ \texttt{on}\ \psi,$$

  where $p \in \mathbb{R}$ and ψ is an XPath identifying query. For k local constraints with queries $\psi_1, \ldots, \psi_k$, the set of modifiable values $\mathcal{M}(d)$ is the set of all identified elements of $\psi_1, \ldots, \psi_k$, i.e. $\mathcal{M}(d) = \bigcup_{i=1}^{k} \mathcal{I}(d, \psi_i)$.

  Recalling that a watermarked document $d_{\vec{w}}$ is such that, for all $i \in \mathcal{M}(d)$,

  $$\vec{v}(d_{\vec{w}})[i] = \vec{v}(d)[i] + \vec{w}[i],$$

  document $d_{\vec{w}}$ is said to respect the local constraint Γ if and only if $\forall i \in \mathcal{I}(d, \psi), |\vec{w}[i]| \leq p$.

- A `global` constraint has the following form:

  $$\Gamma \equiv \texttt{global}\ p\ \texttt{on}\ \psi,$$

  where $p \in \mathbb{R}$ but now ψ is any query returning a numerical value. This query may be expressed e.g. in the XQuery language as in our example. A document $d_{\vec{w}}$ is said to respect the global constraint Γ if and only if $|\psi(d) - \psi(d_{\vec{w}})| < p$. Hence this property is a global property of the watermark $\vec{w}$, not a local property of a unique $\vec{w}[i]$.

- An `invariant` constraint is defined as follows:

$$\Gamma \equiv \texttt{invariant } (\$a_1 \texttt{ in } \varphi_1, \ldots, \$a_k \texttt{ in } \varphi_k) \texttt{ where } \psi.$$

  Queries $\varphi_1, \ldots, \varphi_k$ are XPath queries. Query ψ is any boolean query on parameters $\$a_1, \ldots, \$a_k$. We define the set $E(d, \psi)$ of tuples in $\mathcal{I}(d, \varphi_1) \times \cdots \times \mathcal{I}(d, \varphi_k)$ that verify ψ. A watermarked document $d_{\vec{w}}$ satisfies this constraint if $E(d, \psi) = E(d_{\vec{w}}, \psi)$.

- A `check` *program* clause represents a call to an external program that checks constraints (e.g. a computation not easily definable in XQuery). This clause is respected by document $d_{\vec{w}}$ if the *program* answers "yes" with d and $d_{\vec{w}}$ as input. This corresponds to the usability plugins of [15].

- The semantic of the obvious `linear` constraint is not given, due to the lack of space.

Finding alterations of the document that respect such constraints may be a difficult computational task. The next section shows how to detect interesting patterns in these constraints, in order to optimize the watermarking process.

4 Fingerprinting as an optimization problem (first approach)

4.1 Overview

Fingerprinting If we want to prove not only ownership on the document, but also that a given purchaser is the one that performs unauthorized copies, we should distribute several distinct watermarked versions of the document. This latter technique is known as fingerprinting. In this new setting, we do not want only one watermarked document, but several, according to the number of servers we want to populate. Observe that, to do so, one has now to find several good and distinct watermarks, which may be computationnaly intensive.

Overview of the optimization method First, we consider `check` constraints. Since they are only defined as an external oracle, there is no hope of finding any heuristic to gain computing time. In that case, the greedy method [15] presented in the introduction is probably the most suitable. We will refer to this method as the following function:

$$\texttt{GreedyMark(message } m\texttt{,identifiers } \mathcal{M}\texttt{,secret } \mathcal{K}\texttt{,constraints } \mathcal{L}\texttt{,}\mathcal{G}\texttt{)}.$$

This function returns a watermark $\vec{w}$ that hides message m into identified modifiable values pointed by $\mathcal{M}$, while respecting constraints in $\mathcal{L}$ and $\mathcal{G}$. The encoding uses the secret key $\mathcal{K}$.

For the remaining constraints, we are searching for patterns that can be translated into linear constraints. Hence we split the set $\mathcal{G}$ of usability constraints into two sets: the set Lin of linear constraints, and the remaining set Gen of general constraints. We will first resolve usability constraints from Lin, obtaining a partial instantiation of a good watermark vector, say $\vec{w}_1$. Watermark positions left undefined will be denoted by $\mathcal{M}/\vec{w}_1$. Finally, constraints from Gen will then be explored using the greedy method `GreedyMark` on positions $\mathcal{M}/\vec{w}_1$, obtaining a complete watermark vector $\vec{w}$.

4.2 Translation into linear constraints

Example Translation of constraint C_9 is immediate. Constraints C_1 to C_5 of our previous example result in the following ILP system on document `mills.xml`:

$$
\begin{array}{lll}
-10 \leq & \vec{w}[\mathtt{x}_{\mathtt{Bretagne}}] & \leq 10 \quad \mathtt{C1} \\
-10 \leq & \vec{w}[\mathtt{y}_{\mathtt{Bretagne}}] & \leq 10 \quad \mathtt{C1} \\
-10 \leq & \vec{w}[\mathtt{x}_{\mathtt{Vendee}}] & \leq 10 \quad \mathtt{C1} \\
-10 \leq & \vec{w}[\mathtt{y}_{\mathtt{Vendee}}] & \leq 10 \quad \mathtt{C1} \\
-1 \leq & \vec{w}[\mathtt{height}_{\mathtt{Bretagne}}] & \leq 1 \quad \mathtt{C2} \\
-1 \leq & \vec{w}[\mathtt{height}_{\mathtt{Vendee}}] & \leq 1 \quad \mathtt{C2} \\
-20 \leq & \vec{w}[\mathtt{prod}_{\mathtt{Bretagne}}] & \leq 20 \quad \mathtt{C3} \\
-20 \leq & \vec{w}[\mathtt{prod}_{\mathtt{Vendee}}] & \leq 20 \quad \mathtt{C3} \\
-20 \leq & \vec{w}[\mathtt{prod}_{(55,22)}] & \leq 20 \quad \mathtt{C3} \\
-20 \leq & \vec{w}[\mathtt{prod}_{(22,51)}] & \leq 20 \quad \mathtt{C3} \\
 & & \\
-10 \leq & \vec{w}[\mathtt{prod}_{\mathtt{Bretagne}}] + \vec{w}[\mathtt{prod}_{\mathtt{Vendee}}] & \quad \mathtt{C4} \\
 & +\vec{w}[\mathtt{prod}_{(55,22)}] + \vec{w}[\mathtt{prod}_{(22,51)}] & \leq 10 \\
 & 53 + \vec{w}[\mathtt{prod}_{(22,51)}] & < 60 \quad \mathtt{C5} \\
 & 0.304 * \vec{w}[\mathtt{height}_{\mathtt{Vendee}}] + \vec{w}[\mathtt{height}_{(55,22)}] & \leq 30 \quad \mathtt{C9}
\end{array}
$$

Constraints C_6 and C_8 can not be linearized: there is no way to know if C_8 has a linear counterpart, and C_6 needs squaring of values. Observe also that C_7 can be linearized (as explained in the sequel), but its conditions do not hold in the original document.

Automatic translation Based on the previous example, we identify a set of constraint patterns that can be directly translated into a linear program. Theses patterns express useful usability constraints on the data and can be easily recognized.

We consider four patterns: local constraints, weight-independent sum constraints, cluster constraints and join constraints. For each pattern, we give its general syntactic form, specific restrictions that must be checked, and its translation into a linear inequation.

1. Local constraints (e.g. C_1, C_2, C_3)
   - Pattern: $\Gamma \equiv$ `local` p `on` ψ
   - Restrictions: none
   - ILP constraint: $\forall i \in \psi(d), -c \leq \vec{w}[i] \leq c$
2. Weight-independent sum constraints (wis-constraints): a constraint is said to be *weight-independent* if the set of value identifiers involved in the query computation is the same, whatever the perturbations are on identified values.

   For example, constraint C_4 is weight-independent: even if we modify the `prod` values, the set of `windmill` or `mill` that are involved does not change. This property allows for computing once and for all the set of identified values used in a query computation, and for linking variables to these values into the linear system.

   The weight-independence property is fulfilled by any constraint that do not use modifiable elements in its conditions. This can be statically verified. The formal pattern is the following:

- Pattern: $\Gamma \equiv$ `global` p `on` ψ
- Restrictions: ψ has the following pattern:

  `for (`$\$a$ `in` φ_1`)`
  `return sum(`$\$a/\varphi_2$`),`

  where φ_1 is an XPath query *that do not involve modifiable elements*, and φ_2 is an XPath query that defines modifiable elements.
- ILP constraint:

$$-p \leq \sum_{i \in \mathcal{I}(d,\varphi_1/\varphi_2)} \vec{w}[i] \leq p.$$

An example of weight-*dependent* constraint is given below:

```
global 10 on (
   for $a in /mills/windmill[prod<100]
   return sum($a/prod)
).
```

If a `prod` is equal to 99, watermarking it to 100 will exclude it of the previous linear encoding.

The weight-independent sum pattern can be easily extended to handle the `mean` aggregate. However, the useful statistical `variance` operator can not be handled, as it is not linear.

3. Cluster constraints (for example C_5)
   - Pattern: $\Gamma \equiv$ `invariant` ($\$a$ `in` φ) `where` $\$a/\psi$ θc.
   - Restriction: ψ is an XPath query pointing to a modifiable element, $\theta \in \{=, <, >\}$ and $c \in \mathbb{R}$.
   - ILP constraint: for all element i in $\mathcal{I}(d, \varphi)$:

$$\vec{w}[i] + \vec{v}(d)[i] \ \theta \ c.$$

4. Weight-independent join constraints (for example C_7)
   - Pattern: $\Gamma \equiv$ `invariant` ($\$a_1$ `in` $\varphi_1, \$a_2$ `in` φ_2)
     `where` $\$a_1/\psi_1 = \$a_2/\psi_2$.
   - Restriction: ψ_1 and ψ_2 are XPath expressions defining modifiable elements, φ_1, φ_2 are XPath expressions *that do not depend on any modifiable values.*
   - ILP constraint: for any pair of elements (i, j) in the resultset of the join defined by $(\$a_1, \$a_2)$, we add the constraint:

$$\vec{w}[i] = \vec{w}[j].$$

4.3 The algorithm

Clearly, any watermark satisfying the linear system respects all *Lin* constraints. Our aim is then to extend Agrawal and Kiernan's algorithm [4] so that only good watermarks are selected. The sketch of the resulting algorithm is as follows:

- We compute the distortion $\Delta[i]$ that Agrawal and Kiernan's method would have chosen for an element i;
- We create a new integer (0,1)-variable $\vec{s}[i]$ in the linear system for each i, meaning that element i is a good candidate for watermarking while preserving constraints;
- We force the watermark $\vec{w}[i]$ to be 0 if $\vec{s}[i] = 0$, and $\Delta[i]$ otherwise (this is expressible in a linear system since $\Delta[i]$ is a constant);
- We choose the watermark that maximizes the number of $\vec{s}[i]$ equals to 1, i.e. we solve an integer linear program.

For detection, we look at positions i where $\vec{s}[i] = 1$, extract the corresponding bits and compute the correlation with the hidden bits. The overall algorithm is depicted in Figure 4. It includes Li, Swarup and Jajodia 's extension [12] to fingerprinting of the initial Agrawal and Kiernan's [4] algorithm, using a majority voting (this procedure, `thresholdMajority`, returns the word formed by bits with the highest vote. We do not include it here.) Symbol $S_t(k)$ denotes the output number t of a pseudo-random generator seeded by k (see [4].)

Blindness This algorithm is blind in the sense of [15]: it does not require the original dataset for detection. But, as explained in [15], positions used for a constraint-preserving watermarking *must* be recorded for future detection (here, positions i where $\vec{s}[i] = 1$). This is not a limitation since this set can be efficiently compressed (by e.g. simple interval encoding).

Robustness It should be observed that, when no usability constraints are to be preserved, this algorithm yields exactly the same watermark as Agrawal and Kiernan's (i.e. all $\vec{s}[i]$ equal 1). Hence, its robustness against attacks is the same. When considering usability constraints, the robustness depends on the intrication of these constraints. A too complex group of constraints may yield a non-watermarkable dataset, but experimental evaluations shows that on practical constraints, watermarks are still available (see section 7.)

Generality of the method In our example, constraints C_6 and C_8 do not have a direct characterization in terms of linear constraints. But, from the theoretical point of view, it should be noted that Integer Programming is NP-complete. Hence the method potentially applies to any set of constraints in NP, though not in a straightforward manner.

Problem reduction State-of-the-art ILP solvers (like Ilog Cplex, Dash Xpress-Mp, IBM OSL, etc.) can handle classically up to 10^4 variables. If the number of modifiable values exceeds this limit, which is likely to occur on large datasets, several methods can be used:

- apply standard reduction techniques to lower the number of useful variables [14, 19];
- work only on active identifiers, i.e. those used in query evaluation;
- choose a random subset of variables, or group them according to a secret.

```
LinearMark(document d, message m, constraints 𝓛, Lin, Gen,
      parameters 𝒦, ξ, γ)

      𝒫 :=empty linear program
      foreach element i in 𝓜(d)
         if(S1(i ∘ 𝒦) mod γ = 0) // try mark this element
           j :=S2(i ∘ 𝒦) mod ξ // bit index
           k :=S3(i ∘ 𝒦) mod |m| // letter index

           mask :=S4(i ∘ 𝒦) mod 2;
           mark :=m[k] ⊕ mask // mark bit

           mvalue :=v⃗[i];
           mvalue[j] :=mark
           Δ[i] :=(v⃗[i] − mvalue)
           // compute distortion ∈ {−2^j, 0, 2^j}
           add linear constraint to 𝒫:
              (0 ≤ s[i] ≤ 1, integer)
              (w⃗[i] = Δ[i].s[i])

      add translation of Lin to 𝒫
      w⃗1 :=solve(𝒫, max_s⃗(#s_i = 1))
      w⃗ :=GreedyMark(m, 𝓜/w⃗1, 𝒦, 𝓛, Gen)
      return (v⃗ + w⃗, s⃗)

LinearDetect(suspect document d, s⃗, word size l, key 𝒦, ξ)

      for all p in {1, ..., l}
         vote[p][0] :=0
         vote[p][1] :=0

      for each element i in d appearing in s⃗
         j :=S2(i ∘ 𝒦) mod ξ // bit index
         k :=S3(i ∘ 𝒦) mod l // letter index
         mask :=S4(i ∘ 𝒦) mod 2 // mask bit

         readMark :=v⃗[i][j] ⊕ mask // read the mark

         vote[k][readMark] :=vote[k][readMark]+1
         // voting

      return thresholdMajority(vote)
```

Figure 4: Watermarking with ILP (first approach)

5 Fingerprinting as an optimization problem (second approach)

Pairing algorithm In this section we develop a construct that allows to find not one good watermark, but a large number of good watermarks, by analyzing the form of constraints. We will obtain a set of l watermark positions where *all* binary messages of size l can be encoded. This means that for any message, their encoding will necessarily lead to good watermarks that respects usability constraints. The main point is that these positions are to be computed *only once*. No other usability constraint checking is needed for further message encodings (for constraints respecting our patterns).

The difference with the previous algorithm is twofold. First, the pairing algorithm does not consider all possible valid watermarks, but focus on a restricted family. Thus, fewer watermarking bits can be discovered, but without solving an integer linear program, hence saving computing time. Second, its algorithm can be easily deployed in external memory, which allows for a better scalability.

Let $\mathcal{I}(d, \psi)$ be the set of identifiers whose value is used in the computation of a query ψ. Observe that if ψ is a weight-independent sum constraints, this set $\mathcal{I}(d, \psi)$ is the same, on the original instance and on the watermarked one (since identifiers used in this query are not affected by value modifications). Hence the distortion induced by the addition of watermark $\vec{w}$ is only the sum of marks $\vec{w}[i]$ whose $i \in \mathcal{I}(d, \psi)$.

We illustrate the algorithm on our example, for the following weight-independent sum constraints:

$\psi_1 \equiv$
```
global 0 on (
    for $a in mills/windmill
    return sum($a/prod))
```
$\psi_2 \equiv$
```
global 0 on (
    for $a in mills/(windmill|mill)
    return sum($a/prod)).
```

These constraints mean that the total production on all mills and windmills should be preserved, and also the specific total production for windmills only.

In this setting, identified productions correspond to identifiers $Bretagne$, $Vendee$, $(55, 22)$ and $(22, 51)$ (recall that mills elements are identified by their positions). Values associated with these identifiers are $125, 223, 443, 53$. Query ψ_1 has $125 + 223$ as a result, hence depends on `prod` from $Bretagne$ and $Vendee$. Query ψ_2 has $125 + 223 + 443 + 53$ as a result, and depends on `prod` from `Bretagne`, `Vendee`, $(55, 22)$ and $(22, 51)$. We represent this information in the following *dependency matrix* $A(\psi_1, \psi_2)$:

	$Bretagne$	$Vendee$	$(55, 22)$	$(22, 51)$
ψ_1	1	1	0	0
ψ_2	1	1	1	1

Let $i_1, \ldots, i_{2l}$ be the set of available identified values. The aim of the pairing algorithm is to partition these values *into l dependency pairs* $\{(i_1^1, i_1^2), \ldots, (i_l^1, i_l^2))\}$, so that, for all $j \in \{1, \ldots, l\}$,

- i_j^1 and i_j^2 are involved in the *same constraints*;
- watermark distortions on i_j^1 and i_j^2 will be opposite.

Going back to our example, productions of `Bretagne` and `Vendee` are involved in $\{\psi_1, \psi_2\}$. Productions of `(55,22)` and `(22,51)` impact on $\{\psi_2\}$ only. Hence the first pair will be (`Bretagne`, `Vendee`) and the second ((`55`, `22`), (`22`, `51`)). When hiding message "10" for example, the corresponding mark could be:

- $+1$ on the prod of Bretagne and -1 on Vendee for the first pair;
- -1 on (55,22) and $+1$ on (22,51) for the second pair.

Observe that the overall distortion on constraints ψ_1 and ψ_2 would always be 0.

Ensuring blindness In order to produce a blind algorithm, the alteration on a pair (i^1, i^2) will only depend on the key of i^1. According to this key and the allowed local distortion, we secretely choose a bit position *index*. If the numerical values $\vec{v}[i^1]$ and $\vec{v}[i^2]$ are equal on this bit position, we can not use the pair for watermarking. On the contrary, if they differ on this position, we can permute these bits without altering usability constraints. We act as follows:

- We choose a secret binary mask *mask*;
- To encode a '1', we put $(1 \oplus mask)$ on $\vec{v}[i^1][index]$ and $(0 \oplus mask)$ on $\vec{v}[i^2][index]$;
- To encode a '0', we put $(0 \oplus mask)$ on $\vec{v}[i^1][index]$ and $(1 \oplus mask)$ on $\vec{v}[i^2][index]$.

Since these bits were different in the original dataset, this operation does not change their sum, and the contribution of this pair to the global distortion is still zero. The complete algorithm is presented on Figure 5. It should be observed that, as previous query-preserving algorithms [15], the set of positions used for watermarking must be recorded for further detection. Anyway, this set allows for an efficient compact representation.

`ComputePairs`: efficient pairs computation The core of the method is the pairs computation, hence its implementation must be optimized to achieve scalability. In our prototype, this task is mainly devolved to the DBMS by the following mapping. Given n constraints $\psi_1, \ldots, \psi_n$, we populate a table $matrix(i, f_1, \ldots, f_n)$, such that $f_k = 1$ for i if the identified value i is involved in constraint ψ_k, and zero otherwise (this construction can be done using n `update` queries). We then iteratively traverse the table *matrix ordered by values of* $(f_1, \ldots, f_n)$, using a cursor. This way, identifiers that impact exactly the same constraints are grouped. Two steps of the cursor yield almost surely a dependency pair. To enforce security, identified values are first sorted according to a secret order, known only by the legitimate owner. Hence the exact chosen pairing is secret.

Matrix reduction and non-zero constraints It is possible to reduce the number of lines of the dependency matrix. Observe first that if ψ_1 and ψ_2 depends on *exactly* the same values, it is sufficient to use ψ_1 in the dependency matrix, without changing the solution.

Second, this technique fits well for zero distortion constraints. For the sake of simplicity, suppose that we watermark only with marks +1 or -1, and that all constraints have the same global distortion t. Hence, if two queries ψ_1 and ψ_2 depend on the same values except on t positions, using only ψ_1 in the dependency matrix may introduce a maximal distortion of at most t on query ψ_2. For the general case, we delete all queries that are identical up to t divided by the maximal allowed local distortion on each element.

```
PairMark(document d,message m, wis-constraints C,
        parameters K, ξ, γ)

        pairs=ComputePairs(d,C,K)
        // pairs of equal classes, with a pseudo-random order
        // This computation is done only once.

        for each pair (i_1, i_2) in pairs
           if(S_1(i_1 ∘ K) mod γ = 0) // try to mark this pair
             j:=S_2(i_1 ∘ K) mod ξ // bit index
             k:=S_3(i_1 ∘ K) mod |m| // letter index

             if(v[i_1][j] ≠ v[i_2][j]) // bits 1-0 or 0-1
               mask:=S_4(i_1 ∘ K) mod 2;
               mark:=m[k] ⊕ mask // mark bit

               v[i_1][j]:=mark
               v[i_2][j]:=(not mark)

               add (i_1, i_2) to markList

        return markList

ComputePairs(document d, wis-constraints {ψ_1, ..., ψ_k}, K)
        for each i ∈ M(d)
           set matrix(i, f_1, ..., f_k) to (i, 0, ..., 0)
        for each ψ_j
           for each i used by ψ_j // compute ψ_j
             set f_j to 1 in matrix(i, f_1, ..., f_k)
        sort matrix according to i, using secret order(K)
        then sort according to (f_1, ..., f_k)
        set cursor to the beginning of matrix
        repeat
           i_1=next(matrix)
           i_2=next(matrix)
           if(i_1 = i_2) on (f_1, ..., f_k) // same dependency ?
             add (i_1, i_2) into pairs
        until (end of matrix)
        return pairs

PairDetect(suspect document d,markList,word size l,key K,ξ)

        for all p in {1, ..., l}
           vote[p][0]:=0
           vote[p][1]:=0

        for each pair(i_1, i_2) in markList
           j:=S_2(i_1 ∘ K) mod ξ // bit index
           k:=S_3(i_1 ∘ K) mod l // letter index
           mask:=S_4(i_1 ∘ K) mod 2 // mask bit

           readMark:=v[i_1][j] ⊕ mask // read the mark

           vote[k][readMark]:=vote[k][readMark]+1
           // voting

        return thresholdMajority(vote)
```

Figure 5: Pairing algorithm (second approach)

Finally, one may observe that, instead of solving a integer linear system $A.\vec{w} \leq \vec{c}$ as proposed in section 4, the pairing algorithm builds a restricted basis of the kernel $A.\vec{w} = 0$.

Capacity One may wonder if we will really find such pairs with equal classes, or if too many different classes will appear. Theoretical arguments [8] show that we are likely to find such pairs. We assess this property by our experiments in section 7.

Handling join constraints This technique can be extended also to take into account join constraints. For a condition $\vec{w}[i] = \vec{w}[j]$, we suppress j from the set of modifiable identifiers $\mathcal{M}$. When a value is assigned to $\vec{w}[i]$, we propagate it to $\vec{w}[j]$.

Handling cluster constraints We simply reduce the allowed distortion on each identified value in a cluster constraint.

Robustness An attacker that performs random alterations is more likely to destroy a pair than a single position. Hence a watermark bit built on a pair is less robust than a bit that uses only a single position. This is the price to pay to obtain the computation speed-up and the collusion security explained in the following section. But this phenomenon is balanced by the large amount of pairs discovered by the pairing algorithm. This allows for a large repetition of the bit encoding, which enforces robustness. Experimental evaluations (section 7) assess this property.

If an attacker knows which usability constraints are preserved in his watermarked dataset, a more sophisticated attack may be ruled. By running the `ComputePairs` algorithm (which is naturally assumed to be public), the attacker may find the dependency of identified values. but although the attacker knows that pairs are chosen to have the same dependency, no other information on the exact pairing leaks. Indeed, the pairing is chosen according to a secret order, known only by the data owner.

Summary In the following table we sum up the number of query computations needed to find T distinct watermarks. Parameter n_l denotes the number of linearizable constraints and n_g the number of non-linear constraints. Remember that each query must be computed on a likely huge number of tuples.

Method	#(query computation)	#(ILP solving)
Greedy	$T.(n_g + n_l)$	0
Linear	$n_l + T.n_g$	T
Pairing	$n_l + T.n_g$	0

6 Collusion-secure fingerprinting

Example In the previous sections we focused on simple fingerprinting, i.e. giving to each client a different watermarked version. But in a strong adversarial setting, a collusion of malicious clients can compare their watermarked databases, detect the set of positions where they differ, and replace these positions by a different value. Note that finding a good value is not an easy task for the malicious clients, since they should respect global and local distortions. But this gives them a serious hint to attack the watermark.

Suppose that we want to sell a document to three clients c_1, c_2 and c_3. In order to track back to a malicious purchaser, we will actually distribute distinct watermarked versions, with different embedded messages m_1, m_2 and m_3. When a suspect document is discovered with e.g. the message m_1 in it, we can consider c_1 as guilty. But c_1 may be a victim of clients c_2 and c_3, who actually built the suspect document by a collusion attack. Client c_1 has been framed by c_2 and c_3.

This issue has been extensively studied, and frameproof and collusion secure codes have been designed to resist such attacks [5, 9, 16]. The idea is to produce watermarks by encoding a message m_j for each client j, chosen from a well designed *codebook* $\Gamma = \{m_1, \ldots, m_T\}$.

For our example, the following codebook is frameproof: $\Gamma = \{100, 010, 001\}$. Suppose that bits are encoded on positions i_1, i_2 and i_3. Observing their documents, c_2 and c_3 could not see a difference on their document on position i_1, so they would not modify bits in this position. Hence, the codeword for c_1 can not be produced by this coalition, and c_1 can not be framed.

The important point for this codebook to work is that watermarks must by encoded on *fixed positions*, since these positions are used to detect differences. Hence one should be able to find several good watermarks that distort on the *same positions*. The same requirement holds for collusion-secure codes.

Greedy method The greedy method does not meet this requirement. It may hide the correct codebook message "100" for client c_1 on positions, say, i_5 to i_7. It may also succeed in encoding the codewords for c_2 and c_3 on positions i_8 to i_{10}, i.e. completely different positions. Hence documents of c_1 and c_2 differ on *all* watermarked positions, and all bits of c_1 are exposed. Thus, although we chose good messages, the fixed position requirement of these codebooks is not satisfied by the greedy method.

A natural enhancement of the greedy method is as follows. For the first codeword m_1 to encode, we record the found positions $\{i_1, \ldots, i_n\}$. For the remaining codewords, we force their encoding on positions $\{i_1, \ldots, i_n\}$. If usability constraints are not checked, we discard the codeword and try another one. This gives good watermarks according to the collusion secure fingerprinting principle, but the code rate may be smaller than the one guaranteed for the code itself.

Using the pairing algorithm We would like to pinpoint that encoding messages in pairs obtained from our pairing algorithm leads always to good watermarks. If a large number of such pairs has been discovered, then a good strategy is to use these solutions as a seed for the greedy algorithm, since we are sure to walk around a subset of good watermarks. This gives the algorithm of Figure 6.

7 Experimental results

Context The previous methods are implemented into our open platform `Watermill` [1]. The following experiments were performed on a Dell Latitude D600 laptop with an Intel Pentium M 1.7 GHz - 2 MB L2 cache CPU, 512 MB RAM, and a 5400 RPM hard disk. The system is a Ubuntu 4.10 GNU/Linux standard installation, with Sun JDK 1.4.2, Postgresql 7.4.5 with no special tuning. Swap space is 1 GB.

Benchmark documents Experiments were performed on two different benchmarks: the Forest CoverType database from the UCI KDD archive [2] and a synthetic benchmark.

```
Fingerprint(document d, codebook Γ,client number n,
  constraints L, Lin, Gen, wis-constraints C)

  repeat
        choose an unused codeword m_j
        w⃗:=PairMark(d, m_j, C, ...)
  until (L,Lin,Gen satisfied)
  return w⃗
```

Figure 6: Optimized collusion-secure fingerprint computation

- The Forest CoverType database describes several measures on forest parcels, for a total of 581,012 tuples. We have restricted our attention to the `elevation` and `aspect` attributes. The dataset was equipped with virtual primary keys, that do not exist in the original data. We have watermarked the *aspect* attribute, with local distortion 1. We have split the *elevation* values into 50 random overlapping intervals. The 50 corresponding usability constraints impose that the *mean (i.e sum) of aspects* of data with *elevation* in the same interval should *not* be altered by more than 1 unit (hence a 1 global distortion, which is very restrictive[2]).

- A synthetic relational database. We have considered a sales database, with n *products*, each product having a given *cost*. A number of p *shopping carts* are filled with random subsets of k products. We denote such an instance by $B(n, p, k)$. We have considered the following watermarking problem for various values of n, p and k: we would like to watermark the *cost* attribute, so that:

  - the distortion on cost is limited to 1 Euro;
  - the distortion on the total cost for **each shopping cart** is bound to 1.

  Observe that for an increasing number of carts, these constraints are very aggressive and hard to respect simultaneously, even on a small dataset.

All the above constraints are weight-independent, so our technique applies. In the sequel we compare the pairing algorithm (section 5) with the greedy method (a LSB encoding which uses on-the-fly constraint checking as in [15]). We did not consider constraints that are not captured by our framework, since we rely on the greedy method to verify them (hence the computation time would be identical).

Fingerprinting speed and capacity On the Forest CoverType dataset, checking the 50 constraints took about 350 seconds (mean on 10 experiments). For the greedy method, and a watermark size of 581 bits (0.1% of the dataset size), the estimated watermarking time is $(350 \times 581 =)203350$ seconds, i.e. more than 2 days of computation. Using our method, the computation time decreases to about 3 hours, and, in this time period, much more watermark positions were found. Results are summarized in the following table:

[2] the complete set of constraints is given on our Web site [1]

	Greedy method	**Our method**
Hidden bits	581	143872
Watermark density	0.1%	25%
Precomputation (done once)	not required	10h28min.
Obtaining the first watermark	2 days	10h28min + 3h25min.
Obtaining a new watermark	**2 days**	**3h25min.**

Suppose that we want to embed the following message m="`(c)Academic data, client N`", where N is the binary encoding of the purchaser's number. This message has $208 + \log_2(N)$ bits. In order to increase the embedding robustness, we will repeat each bit e.g. 650 times. The number of available bits for N is then $(143872 - 208 * 650)/650 = 13$. Hence, at least 2^{13} distinct purchasers can be identified with this redundancy.

Observe that finding such distinct watermarked documents would require the computation of all usability constraints on 581,012 tuples for each of the 2^{13} clients with the greedy method. On the contrary, the compilation phase of our method required the computation of usability constraints only once on 581,012 tuples. Each watermarked document required *no* other constraints computations.

Going back to our introductory example, if the value time-window of the Forest CoverType dataset was one week, it would not be possible to distribute it to 4 customers using the greedy method, without raising the computing capacity. Using our pairing algorithm would suffice, even on a small desk computer.

On the synthetic dataset, we considered the instance $B(100, p, 3)$ (100 products, p shopping carts each filled with 3 random products) for increasing values of p from 1 to 100. Each experiment was repeated 10 times. The CPU and query evaluation time, the watermarking capacity (number of valid bits found) and watermarking rate (number of watermark bits found per time unit) for the greedy and pairing algorithms are depicted in Figure 7.

In should be observed first that the fingerprinting speed is almost constant for the pairing algorithm, while it is almost linear for the greedy method (since the number of constraints increases with the number of shopping carts). This is not surprising since the pairing algorithm does not check any constraints during fingerprinting.

Second, **as long as computing time does not come into play**, the watermarking capacity of the greedy method is larger than the pairing algorithm. Indeed, the greedy method has the ability to explore a wider part of the watermarking space, while the pairing algorithm is restricted to a specific kind of solution (e.g. looking for *pairs* divides immediately the capacity by two). But **the found watermarks are of a completely different nature**: each set of n positions (pairs) located by the pairing algorithm can encode **any** n-bits message. On the contrary, the greedy method finds a n position watermark for one specific word, and there is no guarantee that the same positions are valid for another one.

Third, when the number of shopping carts increases, the valid watermarking space reduces and both techniques hardly find a solution. It is noteworthy that that the gap between the two techniques decreases.

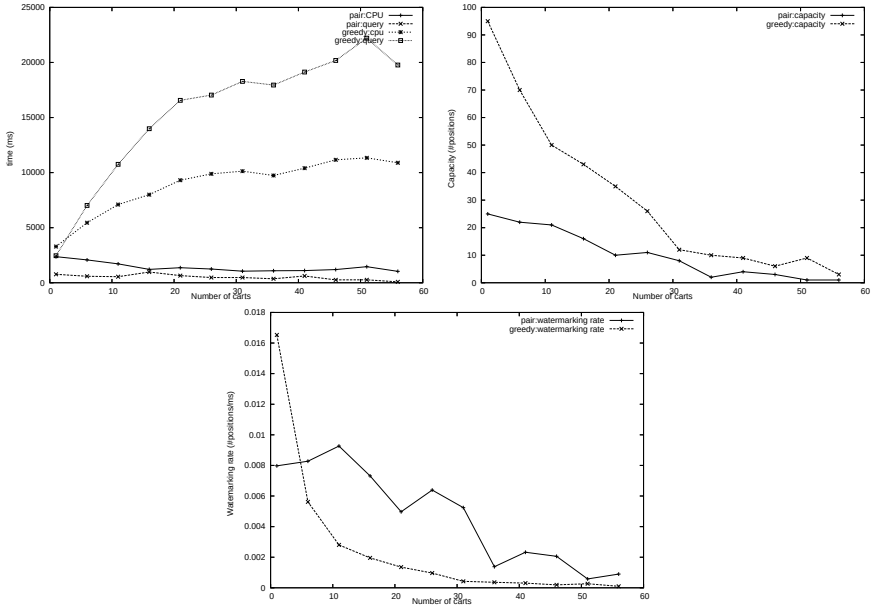

Figure 7: Processing time/watermarking capacity/watermarking rate for the greedy and pairing algorithms

Finally, **if we take computing time into account** by considering the *watermarking rate*, the pairing algorithm appears to **deliver bits faster**.

Collusion-secure fingerprinting We now turn to the robustness of the watermark against a collusion of users. It is well known that collusion-secure codes have a small rate. For example, the code length of Boneh and Shaw's code [5] is $O(c^4 \log(N/\varepsilon) \log(\frac{1}{\varepsilon}))$, where N is the number of messages in the codebook, c is the maximum number of users in a collusion, and ε is the error probability of the detection. On the Forest CoverType benchmark, our method locates $143,872$ available bits. With such a code length, if we consider an error probability $\varepsilon = 10^{-5}$ and that 10% of our purchasers are malicious, we can distribute $N = 50$ distinct documents while resisting to *any* collusion of size at most $c = 5$ with error probability ε. **All these fingerprinted documents will respect the usability constraints**.

Our contribution is on the discovery of large code words that respect usability constraints. But collusion robustness is related to the code itself, so we refer to the literature for a practical evaluation and further elaborations on distribution schemes [11].

Robustness study Due to space limitations, standard robustness study can be found in the appendix.

Conclusion and future work

It this paper, we have presented an optimization technique for the discovery of good watermarks in a dataset that respects several usability constraint patterns. We have also considered the problem of collusion-secure fingerprinting under these constraints.

Natural extensions of this work are the following. First, the number of our constraint patterns could certainly be increased. Second, we would like to address databases fingerprinting where purchasers do not share the same usability constraints. Finally, we would like to devise tools for proving ownership on XML documents after a specific rewriting. For example, one should be able to map identifiable values from the original document into identifiable values in the suspect one.

Acknowledgments

We would like to thank Tahira Jivane, Julien Lafaye and Agnes Plateau for their helpful suggestions and remarks.

References

[1] `http://cedric.cnam.fr/vertigo/tadorne/soft/soft.html`.

[2] `http://kdd.ics.uci.edu/`.

[3] `http://www.eumetsat.de/`.

[4] R. Agrawal, P. J. Haas, and J. Kiernan. Watermarking relational data: framework, algorithms and analysis. *VLDB J.*, 12(2):157–169, 2003.

[5] D. Boneh and J. Shaw. Collusion-secure fingerprinting for digital data. *IEEE Transactions of Information Theory*, 44, 1998.

[6] I. J. Cox, M. L. Miller, and J. A. Bloom. *Digital Watermarking.* Morgan Kaufmann Publishers, Inc., San Francisco, 2001.

[7] W. Fan and J. Siméon. Integrity constraints for XML. In *Symposium on Principles of database systems (PODS)*, pages 23–34, 2000.

[8] D. Gross-Amblard. Query-preserving watermarking of relational databases and XML documents. In *Symposium on Principles of Databases Systems (PODS)*, pages 191–201, 2003.

[9] H.-J. Guth and B. Pfitzmann. Error- and Collusion-Secure Fingerprinting for Digital Data. In A. Pfitzmann, editor, *Information Hiding, Third International Workshop, IH'99*, number 1768 in Lecture Notes in Computer Science, pages 134–145. Springer, 2000.

[10] S. Katzenbeisser and F. A. P. Petitcolas, editors. *Information hiding: techniques for steganography and digital watermarking.* Computer security series. Artech house, 2000.

[11] Y. Li, V. Swarup, and S. Jajodia. Constructing a virtual primary key for fingerprinting relational data. In *Proceedings of the 2003 ACM workshop on Digital rights management*, pages 133–141. ACM Press, 2003.

[12] Y. Li, V. Swarup, and S. Jajodia. A robust watermarking scheme for relational data. In *Proc. 13th Workshop on Information Technology and Systems (WITS 2003)*, pages 195–200, December 2003.

[13] D. Martinenghi. Simplification of integrity constraints with aggregates and arithmetic built-ins. In *Flexible Query Answering Systems (FQAS)*, pages 348–361, 2004.

[14] A. Schrijver. *Theory of linear and integer programming.* Wiley, 1986.

[15] R. Sion, M. Atallah, and S. Prabhakar. Rights protection for relational data. In *International Conference on Management of Data (SIGMOD)*, 2003.

[16] G. Tardos. Optimal probabilistic fingerprint codes. In *Symposium on Theory of Computing*, 2003.

[17] J. Ullman. *Principles of database and knowledge-base systems*, volume I. Computer Science press, 1988.

[18] N. Wagner. Fingerprinting. In *IEEE Symposium on Security and Privacy*, pages 18–22, 1983.

[19] L. A. Wolsey. *Integer Programming.* Wiley-interscience, 1998.

A Robustness study

Data loss attacks We now consider the robustness of our scheme to classical attacks. In the data loss attack, a malicious purchaser suppresses a (large) fragment of data in order to evade detection. We have considered this type of attack on the synthetic benchmark. Figure 8 shows the fraction of recovered mark against destruction of 1 to 100 percent of the dataset.

Data alteration attacks Instead of suppressing data, the purchaser may try to modify the values in a fragment of the data. Notice that modifying these data is likely to break usability constraints. Figure 9 presents the percentage of recovered mark (y-axis) against a progressive data alteration of maximum amplitude 10 (x-axis shows the percentage of altered tuples).

Detection threshold From the previous curves, we can see that choosing $\beta = 0.6$ leads to a robust watermarking scheme against random alteration or random data loss of 50%.

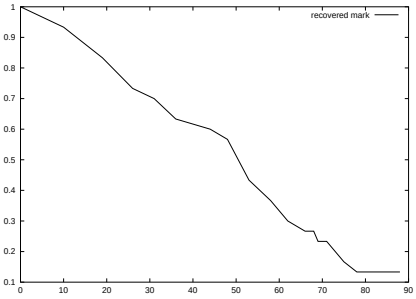

Figure 8: Fraction of recovered bits (y-axis) vs. data loss attacks (x-axis), on the test dataset with 100 products, 6 purchases, and 30 shopping carts

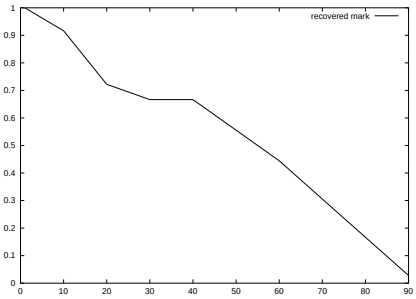

Figure 9: Fraction of recovered bits (y-axis) vs. random alteration with amplitude 10 (x-axis), on the test dataset with 100 products, 6 purchases, and 30 shopping carts