A randomized graph-based scheme for software watermarking*

Lucila Maria Souza Bento[1,2],
Davidson Rodrigo Boccardo[2],
Raphael Carlos Santos Machado[2],
Vinícius Gusmão Pereira de Sá[1],
Jayme Luiz Szwarcfiter[1,2]

[1]Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro, RJ – Brasil

[2]Instituto Nacional de Metrologia, Qualidade e Tecnologia (INMETRO)
Duque de Caxias, RJ – Brasil

lucilabento@ppgi.ufrj.br, {drboccardo,rcmachado}@inmetro.gov.br,
vigusmao@dcc.ufrj.br, jayme@nce.ufrj.br

Abstract. *The insertion of watermarks into proprietary objects is a well-known means of discouraging piracy. It works by embedding into the object some (often surreptitious) data meant to disclose the authorship/ownership of the object. Some promising graph-based watermarking schemes to protect the intellectual property of software have been suggested in the literature, and recent efforts have been endeavored to improve their resilience to attacks. Among the pursued attributes of software watermarking solutions is the one referred to as "diversity", which is the ability to encode the intended information in many distinct forms, making it harder for an attacker to find and remove it. We introduce a graph-based scheme which achieves a high level of diversity through randomization, while admitting an efficient, linear-time implementation nonetheless.*

Resumo. *A inserção de marcas d'água em objetos proprietários é uma conhecida maneira de se desencorajar pirataria. Funciona através da inclusão de alguma informação (em geral escondida) que permita revelar autoria ou propriedade do objeto. Alguns esquemas de marca dágua baseados em grafos para proteger a propriedade intelectual de programas de computador têm sido sugeridos na literatura, e esforços recentes têm sido devotados ao aumento de sua resiliência a ataques. Entre os atributos buscados para soluções de marca d'água de programas está a chamada "diversidade", que é a habilidade de codificar a informação desejada de várias maneiras distintas, tornando mais difícil sua localização e remoção por parte do atacante. Apresentamos um esquema baseado em grafos que consegue, através de randomização, um alto grau de diversidade, permitindo, ainda assim, uma implementação eficiente em tempo linear.*

1. Introduction

For a long time have watermarks been used to enforce authenticity, authorship or ownership of objects. The rationale is that a non-authentic object would not pos-

*Work partially supported by CAPES, CNPq, FAPERJ, Pronametro 52600.017257/2013 and Eletrobrás DR/069/2012.

sess a convincing watermark lookalike, since watermarks are (ideally) hard to be counterfeit. Moreover, a watermarked object would be seriously damaged if one attempted to delete the watermark. In the early 1990's, such ancient idea has been leveraged to the context of software protection as a means to preclude—or at least discourage—the widespread crime of software piracy. A lot of research has been done on software watermarking ever since, and several distinct techniques have been used, including opaque predicates, register allocation, abstract interpretation and dynamic paths [Qu and Potkonjak 1998, Monden and Inoue 2000, Arboit 2002, Nagra and Thomborson 2004, Cousot and Cousot 2004, Collberg et al. 2004].

Graph-based watermarking schemes consist of encoding/decoding algorithms (codecs) that translate the identification data onto (and back from) some special kind of graph. The pioneering graph-based watermark for software protection was formulated in [Davidson and Myhrvold 1996]. It then inspired the publication, in [Venkatesan et al. 2001], of the first watermarking scheme in which the watermark graph is embedded into the *control flow graph* (CFG) of the software to be protected. The CFG, which can be determined by tools for static analysis of code, represents all possible sequences of computation of the program's instructions in the form of a directed graph whose vertices are the blocks of strictly sequential code, and whose edges indicate possible precedence relations between those blocks. The embedder algorithm basically creates dummy code so that new, appropriately interlinked code blocks appear in the CFG, starting at some predefined position and describing exactly the intended watermark structure.

Whereas techniques for watermark embedding are reasonably well developed by now [Collberg and Thomborson 1999, Chroni and Nikolopoulos 2012b, Bento et al. 2013a] and not in the scope of this text , the codecs that have been proposed so far still leave much room from improvement with respect to their resilience to attacks. Two attack models demand special attention, namely *subtractive attacks* and *distortive attacks* [Collberg and Nagra 2009]. In the subtractive attack model, the attacker detects the presence of the watermark and removes it altogether. This kind of attack is basically precluded by code obfuscation and suchlike techniques. The distortive attack model is in a sense more subtle, since the attacker, not being able to detect and remove the watermark as a whole, attempts to damage its structure. It can be done basically by changing the code so that some connections between code blocks disappear (in other words, by indirectly removing edges from its CFG).

The recent, ingenious codec introduced in [Chroni and Nikolopoulos 2012a] was inspired by the work of [Collberg et al. 2003]. It encodes the desired data—which we will refer to as the *key*—as an instance of the reducible permutation graphs introduced by the latter authors. It has been shown in [Bento et al. 2013b] that, even though the watermarks proposed by Chroni and Nikolopoulos manage to withstand attacks in the form of $k \leq 2$ edge removals, there is an infinite number of watermark instances generated by their codec which get irremediably damaged by $k = 3$ edge removals. The recovery of the encoded data is therefore impossible in many cases, even for a modest number of removed edges. In [Chroni and Nikolopoulos 2012c], the authors ask whether graph-based watermarks with greater resilience to attacks—as well as better time/space efficiency—could be devised.

We propose a new codec for software watermarking. The proposed codec employs

Algorithm 1: Encoding the randomized watermark

Input: an integer key ω to be encoded
Output: a randomized watermark encoding ω

1. Let B be the binary representation of ω, and let $n = |B|$. Index the bits of B, from left to right, starting from 1.
2. The watermark $G(V, E)$ is initially isomorphic to a directed path P_{n+1} on vertex set $V = \{1, \ldots, n+1\}$, i.e., the set E initially contains *path edges* from v to $w = v + 1$, denoted $[v \rightarrow w]$, for $1 \leq v \leq n$.
3. For each vertex $v \in V \setminus \{1, n+1\}$, add into E a *back edge* from v to w, denoted $[w \leftarrow v]$, where w is chosen uniformly at random from the elements of V which satisfy:
   - w is not an inner vertex of a cycle of G, and
   - $v - w$ is an *odd* (respectively, *even*) positive integer if v is the index of a bit '1' (respectively, '0') in B.

   If no such w exists, then let v remain with its current outdegree 1, i.e., do not add a back edge leaving v.

randomization to attain a high level of *diversity*, a property whose importance has been noted by the community [Collberg and Nagra 2009], and which is closely related to the resilience of the watermarks against some forms of attack. In short, the structure of the watermarks produced by our scheme is affected by random choices that are made during the execution of the encoding algorithm, which gives rise to a number of distinct graphs encoding the same piece of information. This feature makes it less likely that a watermark can be spotted through brute force comparisons—undertaken by specialized diff tools—among different watermarked programs by the same author or proprietor. Furthermore, the number of edge removals which our watermarks are able to withstand can be customized at will. That is accomplished by means of an edge-to-bit bijection, along with a decoding procedure that is immune to error propagation, making it possible that standard bit-level error-correction techniques are employed in the decoding algorithm quite straightforwardly.

The paper is organized as follows. In Section 2, we introduce the new codec. In Section 3, we propose and analyze a possible linear-time implementation for the encoding and the decoding algorithms. In Section 4, we indicate how to incorporate bit-level error-correction techniques into the new codec. In Section 5, we make our concluding remarks.

2. Randomized watermarks

We introduce a codec for graph-based watermarking of software. The codec has the following main properties:

- the encoding algorithm proceeds in a randomized fashion, therefore the same key will almost certainly give rise to distinct watermarks at different executions of the algorithm;

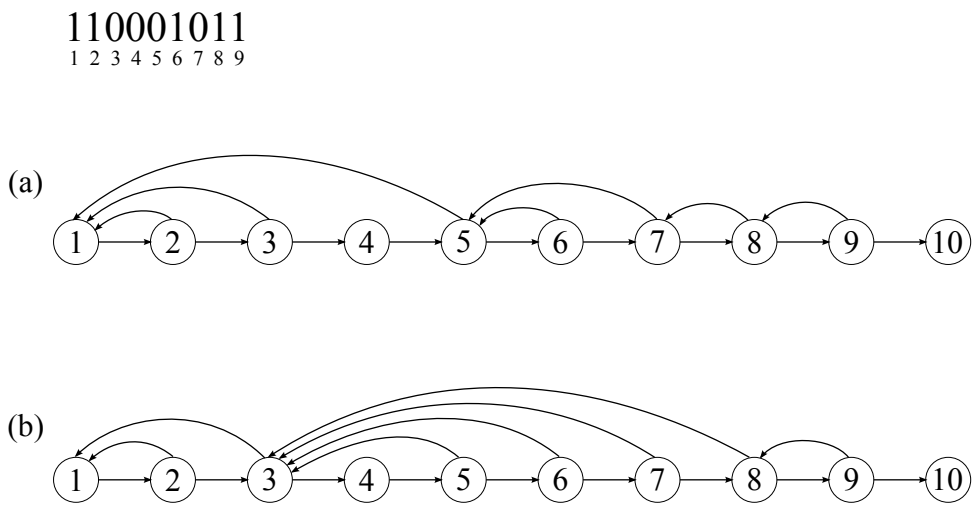

Figure 1. Distinct randomized watermarks encoding the same key $\omega = 395$

- there is a one-to-one correspondence between the edges of the watermark and the bits of the encoded key, hence distortive attacks can be detected *after* the graph-to-key decoding process, and the correction of any flipped bits—up to some predefined number— can be carried out by standard error-correction algorithms;
- both encoding and decoding algorithms can be implemented to run in linear time.

Algorithm 1 describes the basic steps of the encoding algorithm, when no extra bits—intended for error-correction—are used (we address error-correction in Section 4). If $C : v_1, v_2, \ldots, v_d, v_{d+1} = v_1$ is a cycle on d vertices of a directed graph G, we say vertices $v_2, \ldots, v_{d-1}$ are the *inner vertices* of C.

Figure 1 illustrates two watermarks generated by the new codec for key $\omega = 395$, whose binary form, determined in step 1 of the algorithm, is $B = 110001011$, with $n = 9$ bits. Both watermarks have the same number of vertices, namely $n + 1$, and both have a (unique) Hamiltonian path[1], which is created in step 2 of the algorithm. The first vertex of the Hamiltonian path, labeled 1, always corresponds to a bit '1' in B, and its outdegree is always 1. Now, each vertex from 2 to n becomes the origin of either zero or one *back edges*. The back edges with origin in each $v \in \{2, \ldots, n\}$ (or the absence thereof) will bear a one-to-one correspondence with the bits indexed from 2 to n in B: a bit '1' with index $v \geq 2$ in B gives rise to a back edge $[w \leftarrow v]$ constituting an odd-distance "backwards jump" over the Hamiltonian path (i.e., $v - w$ is odd), whereas a bit '0' either gives rise to an even-distance backwards jump or to no jumps at all (when there is no $w < v$ such that $v - w$ is even and w is not an inner vertex of a cycle).[2]

We remark that the event that a back edge cannot be added shall never occur with respect to a vertex $v \geq 2$ corresponding to a bit '1'. Indeed, because vertices are processed left-to-right by the algorithm, vertex $w = v - 1$ is never an inner vertex of a cycle by the time v is processed. Consequently, for all $v \geq 2$ corresponding to a '1', at least the back

[1] A Hamiltonian path on a graph G is a path where all vertices of G appear exactly once.

[2] All arrays in this text are 1-based, i.e. their first position has index 1.

Algorithm 2: Decoding the randomized watermark

Input: a randomized watermark G with $n+1$ vertices
Output: the key ω encoded by G

1. Label the vertices of G in ascending order as they appear in the unique Hamiltonian path of G.
2. Let B be a bit array starting with a bit '1' followed by $n-1$ bits '0'.
3. For each vertex $v \in \{2, n\}$, if there is a vertex $w < v$ such that $[w \leftarrow v] \in E(G)$ and $v - w$ is odd, then $B[v] \leftarrow$ '1'; otherwise, $B[v] \leftarrow$ '0'.
4. Return $\omega = \sum_{i=1}^{n} B[i] \cdot 2^{n-i}$.

edge $[v-1 \leftarrow v]$—an odd-distance backwards jump, as intended—will be available. On the other hand, if a vertex v corresponds to a bit '0' in B, then it is possible that no w can be the destination of a back edge with origin at v constituting an even-distance backwards jump. The absence of a back edge will therefore indicate that the bit with index v in B is a '0'. Moreover, if v gets no outgoing back edges, then vertex $v' = v+1$ is assured to get one, for if v' corresponds to a '0', then at least the edge $[v-2 \leftarrow v]$—an even-distance backwards jump, as intended—will be available, since there is no back edge closing a cycle at $v-1$.

Back to our example in Figure 1, notice that vertex 2 corresponds to a bit '1' in B and therefore receives the outgoing back edge $[1 \leftarrow 2]$, the only possible choice then. Vertex 3, on its turn, corresponds to a '0' and gets to be the origin of back edge $[1 \leftarrow 3]$, again the only possible choice. Now, vertex 4, which corresponds to a '0', must be left without an outgoing edge, for the only $w < 4$ such that $4 - w$ is even would be $w = 2$, but vertex 2 is an inner vertex of the existing cycle $1, 2, 3, 1$. Vertex 5 corresponds to a bit '0', and two back edges were available by the time it was processed, namely $[1 \leftarrow 5]$ and $[3 \leftarrow 5]$. For the watermark in Figure 1(a), the former edge was chosen, whereas the latter was chosen for the watermark in Figure 1(b). The algorithm carries on in similar fashion for vertices $6, \ldots, 9$, and the watermark is complete.

The decoding procedure consists of two steps. First, we must label the vertices of the watermark, so their correspondence to the bits of the encoded binary can be determined. This can always be done, since the blocks of the Hamiltonian path are always consecutive in the CFG, corresponding to vertices $1, 2, \ldots, n+1$. Second, we set the first bit of the binary as '1' (which is always the case, since zeroes to the left of a number are ignored), and we proceed to gathering the information encoded by the back edges, from vertex 2 onwards: a back edge $[w \leftarrow v]$ such that $v - w$ is odd (respectively, even) indicates that the bit with index v in the binary is a '1' (respectively, '0'), and vertices $2 \leq v \leq n$ which are not the origin of a back edge also correspond to bits '0' in the binary. The decoding algorithm is given in pseudocode as Algorithm 2.

3. Linear-time implementation

The two first steps of Algorithm 1 are straightforward. In order to implement step 3, however, we must be able to keep track of the current *destination candidates*, i.e. vertices $w < v$ which are not (yet) inner vertices of any cycles and therefore can (still) be picked as the destination of a back edge with origin at vertex v being currently processed. If v is even and corresponds to a bit '0', or v is odd and corresponds to a bit '1', then the destination w of the back edge $[w \leftarrow v]$ must be selected among the current even-labeled destination candidates; otherwise, w must be selected among the current odd-labeled destination candidates. Whichever the case, the algorithm must choose uniformly at random among the destination candidates whose label has the desired parity, which can be done by picking a random integer between 1 and the number of such candidates.

We employ two stacks, S_0 and S_1, each one implemented over an array so that any item can be looked up by its index in constant time. Implementing those stacks over arrays also allows for a constant-time *pop_all*(i) method, which removes all items whose indexes are greater than a given index i.[3]

The proposed implementation of step 3 consists of a loop that iterates through vertices $2, \ldots, n$ in order to determine the back edges (if any) with origin at each of these vertices, one by one. The following invariant holds: the elements in stack S_0 (respectively, S_1) are precisely the even-labeled (respectively, odd-labeled) destination candidates in ascending order (bottom-up along the stack) at any moment. All vertices $v = 1, \ldots, n$ will be added to their respective stack (even-labeled vertices into S_0, odd-labeled vertices into S_1) exactly once during the execution of the algorithm, namely by the end of the iteration during which v is visited, i.e. right after determining the destination of the back edge with origin in v.

Additionally, we need an auxiliary n-sized array, call it A, which is initially empty, and whose positions are indexed from 1 to n. Each position v of the array, for even v, will be assigned the size that stack S_1 used to have by the time v was added to stack S_0. Analogously, each position v of the array, for odd v, will be assigned the size that stack S_0 used to have by the time v was added to S_1.

Now we can describe a linear-time implementation for the whole step 3 of the encoding algorithm. Its pseudocode is depicted in Procedure 3.

After the initialization of the data structures (line 1), vertex $v = 1$ is the first to be considered. However, since no back edge with origin at vertex 1 is meant to be added, the algorithm just pushes v into S_1 (because v is odd) and writes 0 (the current size of stack S_0) to position 1 of A (line 2).[4] Now, for each vertex $v \in \{2, \ldots, n\}$, the algorithm first decides which stack—S_0 or S_1—contains the candidates among which the destination of the back edge with origin in v shall be (randomly) chosen (lines 4–7). Such stack, which will be referred to as S, will be S_0 if either an even-distance backwards jump is intended ($B[v]$ is a bit '0') and v is even, or an odd-distance backwards jump is intended ($B[v]$ is a '1') and v is odd; otherwise, the appropriate stack will be $S = S_1$. The element in

[3] That can be achieved easily by redirecting a "stack top" pointer to the stack's ith item, or, alternatively, by maintaining a "stack size" variable.

[4] This latter instruction is not actually necessary, since A was initialized with zeroes. We have included it for clarity.

Procedure 3: Determining back edges in $\mathcal{O}(n)$ time

Input: the n-bit binary representation B of the key to be encoded,
and the set E containing only the path edges of the watermark
Output: updated set E containing all (path/back) edges of the watermark

```
1.  S_0 := empty stack;  S_1 := empty stack;  A := array with n zeroes
2.  S_1.push(1);  A[1] := 0
3.  for v = 2, ..., n do
4.      if (v is even and B[v] = '0') or (v is odd and B[v] = '1') then
5.          S := S_0;  S' := S_1
6.      else
7.          S := S_1;  S' := S_0
8.      if S.size > 0 then
9.          j := integer chosen uniformly at random from [1, S.size]
10.         w := S[j]
11.         E := E ∪ {[w ← v]}
12.         S.pop_all(j)
13.         S'.pop_all(A[w])
14.     if v is even then
15.         S_0.push(v);  A[v] := S_1.size
16.     else
17.         S_1.push(v);  A[v] := S_0.size
18. return E
```

$\{S_0, S_1\} \setminus S$ will be referred to as S'. If S is empty, then there are no vertices available to be the destination of a back edge leaving v; in this case, no back edge will be added to the edge set E. Otherwise, an integer j is chosen uniformly at random between 1 and the size of S, thus determining the destination $w = S[j]$ of the back edge leaving v. Such back edge is added to E (line 11). Now, because the addition of a back edge $[w \leftarrow v]$ implies the creation of a cycle $C : w, w+1, \ldots, w+(v-w) = v, w$, all inner vertices of C which used to be destination candidates now become unavailable for future selections. In other words, they cease to be destination candidates, and must therefore be removed from their corresponding stacks, i.e. all vertices $w' > w$ must be popped from both S and S'. Since the index of w in S, call it j, is known, it is easy to remove all such vertices w' from S (line 12), for they are precisely those vertices whose indexes in S are greater than j. On the other hand, because w was never an element of S', there is no such thing as the index of w in S', and one might think that a (binary) search in S' would be called for in order to determine the index of the last element of S' that is smaller than w—which would append an extra $O(\log n)$ factor to the algorithm's time complexity. However, because w is currently *not* the inner vertex of a cycle (otherwise it would not have been selected as a back edge destination), no vertex $w' < w$ was selected as destination after w was processed, which means all vertices that belonged to S' by the time w was processed *still* belong to S', and must remain there. As a matter of fact, only those vertices must remain

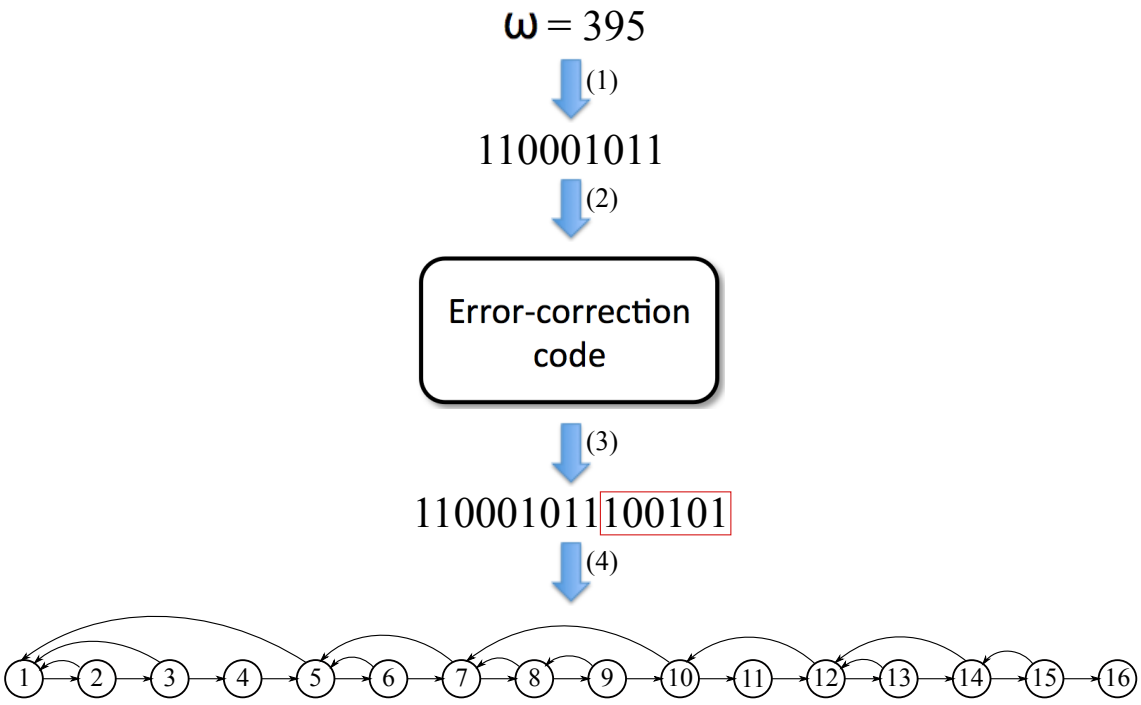

Figure 2. Encoding the key $\omega = 395$ with 1-bit error correction

in S', since vertices added to S' after w was processed are necessarily greater than w (and smaller than v), hence they have just become inner vertices of C and must be removed from S'. In other words, S' must retain precisely its r first elements, where r is the size that S' used to have when w was processed and added to S. And now the auxiliary array A comes into play, for $r = A[w]$ is precisely the value that was stored at its wth position when w was added to S (line 15 or 17, depending on the parity of w).

Since there is a constant number of operations per vertex, and all operations clearly run in $O(1)$ time, the overall complexity of Procedure 3 is $O(n)$.

4. Resilience to distortive attacks

The literature on error detection/correction techniques, mainly intended for binary messages transmitted on error-prone channels, is quite vast [Hamming 1950, Reed and Solomon 1960, Mann 1968, Purser 1995, Wicker 1995]. We do not intend to give an exhaustive account on the existing techniques in this text. Instead, we intentionally regard them as "black boxes", demonstrating how the intended results can be easily achieved. Although the existing error-correction techniques may differ (a lot) in the way they tackle a possibly damaged binary, a common requirement is the insertion of a number $f(n, t)$ of redundancy bits, for some function f. This is done in a preprocessing step of the binary representation of the key about to be encoded.

In the decoding phase of our watermarking scheme, the effect of k malicious edge removals is that of erroneously writing k or less bits '0' at positions originally occupied by bits '1' in the encoded binary. This is so because the absence of a back edge leaving vertex v, for $2 \leq v \leq n$, is regarded by the decoder as a bit '0' with index v. If a back edge leaving v used to exist in the watermark before the attack, then the bit with index v in the original binary might as well be a '1'. Yet, the consequence of each edge removal is

that of a single flipped bit (at most), because, due to the mechanics of the proposed codec, decoding errors *do not propagate*.

Suppose, on the other hand, that, instead of being based on the parity of the distance covered by backwards jumps, our encoding algorithm selected the destination w of a back edge with origin at v in the following way: pick—uniformly at random—a destination $w < v$ such that w is not an inner vertex of a cycle, and w corresponds to a bit, in the binary, that is the same as the bit in the vth position. In other words, if v is a '1', its outgoing back edge must reach a '1'; if v is a '0', its outgoing back edge must reach a '0'. Under such a codec, an edge removal resulting in the erroneous decoding of a vertex v would cascade the error to vertex v' whose outgoing back edge reached v, and to vertex v'' whose outgoing back edge reached v', and so on.

We illustrate, in Figure 2, the encoding of the same key from Figure 1, but now employing the well-known Reed-Solomon error correction technique [Reed and Solomon 1960] under a Galois field GF(2^3), which in this case provides the ability to recover from 1-bit flips (i.e., from single edge removals). In step (1), the binary form of the key is obtained; in steps (2) and (3), the binary is passed to an error-correction preprocessing step, where redundancy bits are appropriately inserted; in step (4), the final binary is translated onto the watermark graph by using the proposed encoding function (see Section 2, Algorithm 1). Notice that the preprocessing step could be made so that an arbitrary, predefined number $t > 0$ of edge flips could be afterwards detected and corrected, yet the size of the ensuing binary grows accordingly.

The decoding is done in similar fashion, as illustrated in Figure 3: in step (1), the watermark graph is decoded into the binary it represents (see Section 2, Algorithm 2); in (2) and (3), the decoded binary is passed to the error-correction post-processing step, wherefrom another binary (with unflipped bits) is produced; and, finally, in step (4), the original key is retrieved.

Thus, if the number k of missing edges is less than the fixed threshold $t > 0$ taken into consideration when preprocessing the original binary, the employed error-correction solution shall identify the flipped bits and correct them; otherwise, the attack will have succeeded in damaging the watermark permanently. In effect, no matter the chosen error-correction technique or the number t of errors the watermark can withstand, the attacker may always remove so large a number $t' > t$ of edges that no recovery is possible, as illustrated in Figure 4.

5. Conclusion

We presented a randomized codec for graph-based software watermarking. Its main property is its ability to encode the same key as distinct graphs, accounting for a high diversity, a feature whose importance has been stressed by the community. Moreover, it can be implemented to run in linear time[5] as shown in Section 3, and it is compatible with standard bit-level error-correction techniques.

For future work, it should be interesting to devise a codec with all these nice attributes, but also with the property of being embeddable in the CFG without the need

[5]An implementation using the Python language is available at `https://www.dropbox.com/s/kydbc60mkl7lf7z/randomized-watermark.py`.

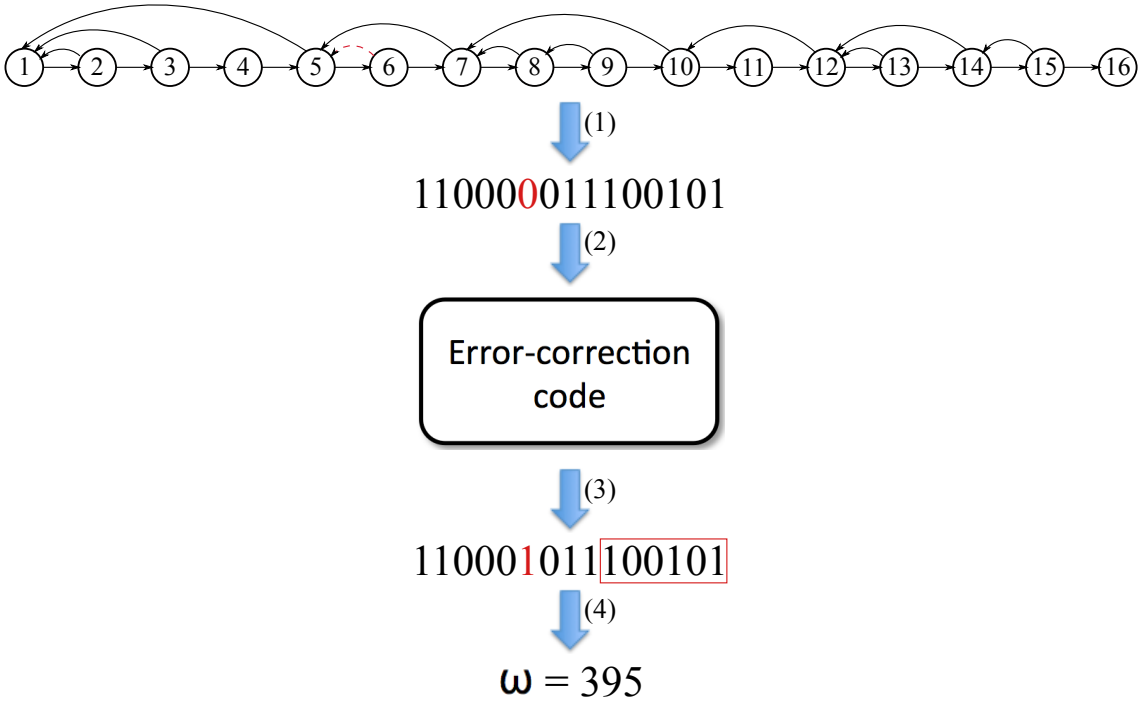

Figure 3. Decoding the watermark after the removal of edge $[5 \leftarrow 6]$

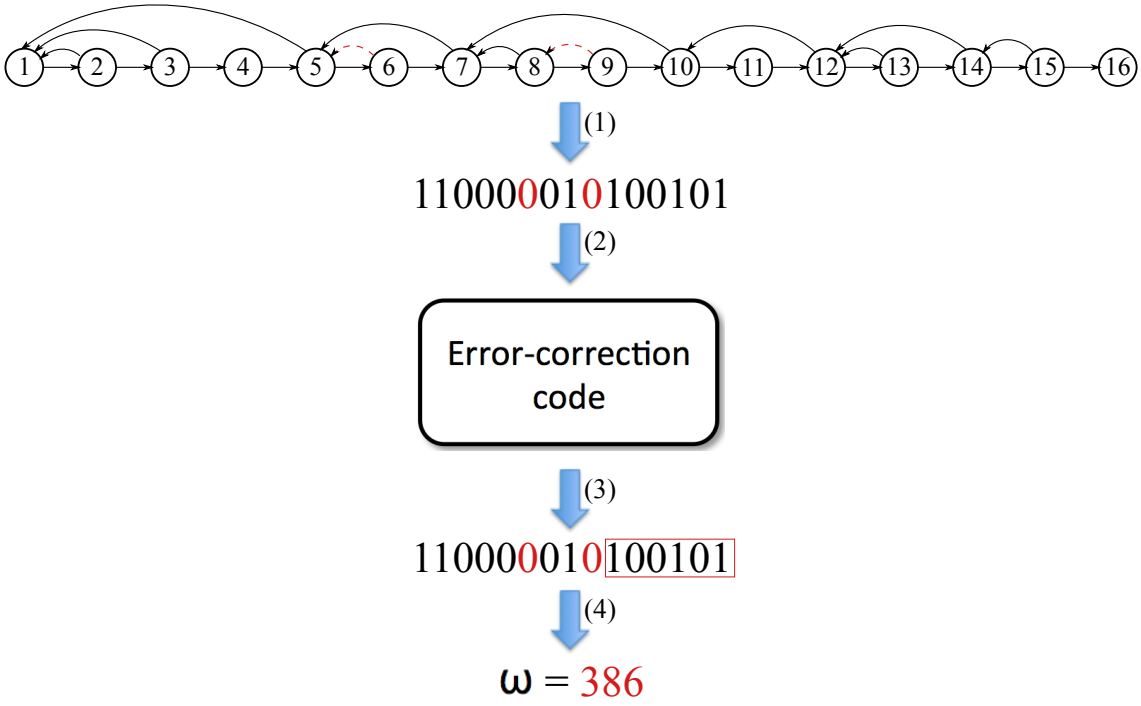

Figure 4. Decoding the watermark after the removal of edges $[5 \leftarrow 6]$ **and** $[8 \leftarrow 9]$**: an incorrect outcome is produced**

to use raw jumps (*goto* statements), something that is required not only by our codec but by all other CFG-embedding codecs we are aware of. In other words, a codec which produces watermarks that can be made to appear as subgraphs of the CFG by means of adding dummy *structured code* only. Such property would improve upon the stealthiness of the produced watermarks, since they would resemble normal, actual code even further.

References

Arboit, G. (2002). A method for watermarking Java programs via opaque predicates. In *Proc. Int. Conf. Electronic Commerce Research (ICECR-5)*.

Bento, L. M. d. S., Boccardo, D., Costa, R., Machado, R. M. S., Pereira de Sá, V. G., and Szwarcfiter, J. L. (2013a). Fingerprinting de software e aplicações à metrologia legal. In *Proc. 10th International Congress on Electrical Metrology (SEMETRO'13)*.

Bento, L. M. S., Boccardo, D. R., Machado, R. C. S., Pereira de Sá, V. G. a. P., and Szwarcfiter, J. L. (2013b). Towards a provably resilient scheme for graph-based watermarking. In *Proc. Workshop on Graph-Theoretic Concepts in Computer Science (WG'13), LNCS 8165*, pages 50–63. Springer.

Chroni, M. and Nikolopoulos, S. D. (2012a). An efficient graph codec system for software watermarking. In *36th IEEE Conference on Computers, Software, and Applications (COMPSAC'12)*, pages 595–600. IEEE Proceedings, 36th edition.

Chroni, M. and Nikolopoulos, S. D. (2012b). An embedding graph-based model for software watermarking. In *Proc. International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'12)*, pages 261–264. IEEE.

Chroni, M. and Nikolopoulos, S. D. (2012c). Multiple encoding of a watermark number into reducible permutation graphs using cotrees. In *CompSysTech*, pages 118–125.

Collberg, C., Carter, E., Debray, S., Huntwork, A., Linn, C., and Stepp, M. (2004). Dynamic path-based software watermarking. In *Proc. Conference on Programming Language Design and Implementation (SIGPLAN'04)*.

Collberg, C., Kobourov, S., Carter, E., and Thomborson, C. (2003). Error-correcting graphs for software watermarking. In *Proc. 29th Workshop on Graph-Theoretic Concepts in Computer Science (WG'03), LNCS 2880*, pages 156–167. Springer.

Collberg, C. and Nagra, J. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.

Collberg, C. and Thomborson, C. (1999). Software watermarking: Models and dynamic embeddings. In *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'99, pages 311–324. ACM.

Cousot, P. and Cousot, R. (2004). An abstract interpretation-based framework for software watermarking. In *Proc. Conference Record of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–185. ACM Press, New York, NY.

Davidson, R. and Myhrvold, N. (1996). Method and system for generating and auditing a signature for a computer program. US Patent 5,559,884.

Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160.

Mann, H. (1968). *Error Correcting Codes*. John Wiley and Sons.

Monden, A. and Inoue, K. (2000). A practical method for watermarking Java programs. In *Proc. 24th Computer Software and Applications Conference*, pages 191–197.

Nagra, J. and Thomborson, C. (2004). Threading software watermarks. In *Proc. 6th International Workshop on Information Hiding, LNCS 3200*, pages 208–233. Springer.

Purser, M. (1995). *Introduction to Error-Correcting Codes*. Artech House Inc.

Qu, G. and Potkonjak, M. (1998). Analysis of watermarking techniques for graph coloring problem. In *ICCAD*, pages 190–193.

Reed, I. S. and Solomon, G. (1960). Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304.

Venkatesan, R., Vazirani, V. V., and Sinha, S. (2001). A graph theoretic approach to software watermarking. In *Proc. 4th International Workshop on Information Hiding (IHW'01)*, pages 157–168. Springer.

Wicker, S. B. (1995). *Error control systems for digital communication and storage*. Prentice Hall.

