Structured Watermarks for Structured Software

Lucila M. S. Bento[1*†], Davidson R. Boccardo[4†], Raphael C.S. Machado[3†], Vinícius G. Pereira de Sá[2†], Jayme L. Szwarcfiter[1,2†]

[1*]State University of Rio de Janeiro, São Francisco Xavier St., Rio de Janeiro, 20550-000, Rio de Janeiro, Brazil.
[2]Federal University of Rio de Janeiro, Athos da Silveira Ramos Ave, Rio de Janeiro, 21941-916, Rio de Janeiro, Brazil.
[3]Fluminense Federal University, Gal. Milton Tavares de Souza Ave, Niterói, 24210-310, Rio de Janeiro, Brazil.
[4]Albert Einstein Israelite Hospital, Brigadeiro Faria Lima Ave, São Paulo, 01451-001, São Paulo, Brazil.

*Corresponding author(s). E-mail(s): lucila.bento@ime.uerj.br;
Contributing authors: davidson.boccardo@einstein.br; raphaelmachado@ic.uff.br; vigusmao@dcc.ufrj.br; jayme@nce.ufrj.br;
†These authors contributed equally to this work.

Abstract

Software watermarking involves integrating an identifier within the software, enabling timely retrieval to disclose authorship/ownership and deter piracy. Various software watermarking schemes have been proposed in the literature, many of which involve statically embedding an encoded identifier into the control flow graph of the program. In this paper, we propose novel embedding and extraction algorithms characterized by four key features: randomization, generating watermarks with a size closely matching the number of bits in the identifier, implementing both encoding and decoding with linear time complexity, and, most importantly, generating watermarks that conform to structured code. We emphasize the capability to encode the same identifier as distinct graphs, coupled with the absence of cumbersome "GOTO"-like substructures, as enhancements to the stealthiness of our watermarks. This feature makes them more resilient to common forms of attack, contributing to their effectiveness in safeguarding software integrity and discouraging unauthorized use.

Keywords: software watermarking, linear-time algorithms, structured code

1 Introduction

Violation of software copyright by means of unauthorized copies is what is known as software piracy. The latest BSA survey [1] revealed that 37% of all software installed on personal computers worldwide were then unlicensed, with an estimated loss that surpassed 46 billion dollars. Another research [2] shows that even the existence of intellectual property protection laws does not significantly reduce piracy. Technical countermeasures are therefore of utmost importance.

Watermarks have been used over the centuries to establish authenticity, authorship or ownership of objects. The concept has been leveraged to the context of software protection. Auditing some illegal copy of a watermarked software artifact may reveal which authentic copy it originated from. Ideally, tampering with an embedded watermark should be impossible.

Software watermarks are generally categorized into two types: static and dynamic [3]. Static watermarks are embedded within a program's code and/or data, while dynamic watermarking techniques store a watermark in a program's execution state. Specific types of static watermarks include Code Replacement, Code Re-Ordering, Register Allocation Based Watermarking, Opaque Predicates, Abstract Interpretation, and Graph-based Watermarking [4].

Graph-based watermarking schemes comprise algorithms (*codecs*) to encode/decode the identifier as/from a graph, and algorithms to somehow embed/extract the encoded identifier into/from the software. The first graph-based watermarking scheme [5] embeds the watermark graph in the Control Flow Graph (CFG) of the software to be protected. The CFG, which can be obtained by static analysis tools, represents all paths that may be followed during the execution of a program. The vertices are strictly sequential code blocks and the edges indicate possible precedence relationships between blocks. The embedder algorithm produces additional lines of code to be inserted in the source code of the program, starting from some predefined, secret position, and making sure not to interfere with the program logic (e.g., dummy lines that will not alter state, or even nonsensical code that is made innocuous by saving memory registers at the beginning and restoring them at the end). The embedded instructions give rise, in the CFG of the program, to a subgraph corresponding exactly to the intended watermark graph.

Despite achieving a reasonable level of development, graph-based watermark codecs proposed thus far exhibit room for improvement in terms of resilience to attacks [6]. Two particularly noteworthy attack models are distortion attacks and subtraction attacks [3]. In the distortion attack model, adversaries aim to compromise the watermark structure by altering the program code, removing specific code blocks or disrupting connections between them. Essentially, the adversary indirectly seeks to eliminate CFG vertices and edges from the marked program. On the other hand, subtraction attacks involve the adversary detecting the watermark and attempting to eradicate it entirely.

Certain graph-based watermark codecs, such as those presented in studies [7, 8], do exhibit a degree of resilience to distortion and subtraction attacks [9–11], particularly when the attacker lacks knowledge of the watermark's location within the CFG. However, while the watermark graphs produced by these codecs are guaranteed to belong

to a subclass of all possible CFGs known as *reducible permutation graphs*[12, 13], they may not necessarily reflect the topology of CFGs in *structured* programs. This is due to some edges requiring the introduction of artificial jumps in a "GOTO" fashion[14]. These artificial jumps are, at the very least, unconventional and may even be impossible in many modern programming languages. Their presence could arouse suspicion in a malicious adversary, potentially increasing the likelihood of identifying watermarks within resulting CFGs and executing attacks.

Our contribution

We propose a linear-time codec for graph-based software watermarking, introducing a novel approach where the code corresponding to the watermark is strictly *structured*, adhering to the formal definition provided by Dijkstra in his seminal paper [14]. Consequently, when safeguarding structured software, the resulting watermarked CFGs will consistently fall within the class of "Dijkstra graphs", as rigorously demonstrated by [15]. This stands in stark contrast to all known previous graph-based watermarking techniques, where the inclusion of (non-structured) "GOTO" statements is unavoidable. Moreover, our encoding algorithm incorporates randomization, ensuring the generation of distinct watermarks for the same identifier across different executions. This innovative approach reduces the likelihood of identifying a watermark by comparing different programs that share watermarks encoding the same identifier, such as those produced by the same author or proprietor. Finally, there is a strict one-to-one correspondence between the (non-path) edges of our watermark graph and the bits of the encoded identifier. As a consequence, not only will our watermarks be remarkably modest in size, but also distortion attacks (whereby the watermark is damaged, but not removed) can be detected after the decoding, making it possible to rectify flipped bits via standard error-correction techniques. This feature distinguishes our approach from other watermark schemes, where the removal of a small number of edges could compromise the entire structure, rendering recovery of the encoded identifier impossible.

Notation

In this paper, we employ standard notation. If G is a directed graph, then $V(G)$ and $E(G)$ denote the set of vertices and edges of G, respectively. For $u, v \in V(G)$, an edge from u to v is represented by $[u \to v]$, or, equivalently, $[v \leftarrow u]$.

Roadmap

Section 2 provides an overview of the previous efforts made in the graph-based software watermarking domain. The proposed encoding and decoding algorithms are given in detail in Section 3, and a thorough encoding walk-through follows in Section 4. Section 6 wraps up the paper and offers possible directions for future works.

2 Software watermarks: evolution and desired properties

In the early 1990s, Davidson and Myhrvold [16] introduced the concept of software watermarking for software protection. Since then, extensive research on software watermarking has been conducted, employing various techniques [6, 17–27].

Collberg and Thomborson [28] defined three types of attack against software watermarks: subtraction attacks, where the attacker removes the watermark from the protected program; addition attacks, where a new watermark is added into the protected program; and distortion attacks, where the attacker modifies the original watermark. The authors also categorized software watermarks as static (stored in the executable program) or dynamic (built during the program execution).

Venkatesan et al. [5] introduced a static graph-based watermark scheme, encoding an integer identifier ω as a graph G. The watermarked program results from modifying the original source code, including the executable binaries, in a way that G becomes a subgraph of the modified program's CFG.

Collberg et al. [29] compiled a list of desired watermark properties:

- Data rate: the size of the watermark should be close to the number of bits of the identifier;
- Performance: dealing with watermarks should not significantly affect the execution time of the program;
- Resilience: a watermarking scheme must be able to recognize a watermark even after an adversary attacks the marked program; and
- Stealthiness: the programs with and without the watermark should look alike, with nearly identical properties, making it hard for one to detect (and tamper with) the watermark.

Chroni et al. [7], inspired by the work of [30], introduced a graph-based watermarking scheme that encoded an identifier in a graph. The proposed scheme converts the binary representation of the identifier into a special permutation, and it converts this permutation into a graph that belongs to a subclass of the reducible flow graphs. Later, the class of graphs produced by their scheme was characterized and called *canonical reducible permutation graphs* [12], where it was also demonstrated that the watermarking scheme proposed by Chroni et al. [7] can withstand attacks in the form of $k \leq 2$ edge removals, and there is an infinite number of watermark instances generated by their codec which get irremediably damaged by $k = 3$ edge removals. A parallel theoretical investigation into the resilience of this watermarking scheme against edge modification attacks in CFG was also conducted by Mpanti et al.[11].

Mpanti et al. [31] investigated experimentally the resilience to edge modification of that same scheme. The results show that attacks modifying k edges, $3 \leq k \leq 7$, are unlikely to make Chroni and Nikolopoulos's schema inadvertently extract and decode a corrupted watermark. It might fail (indicating a problem), but almost surely it will not be fooled. In the stealthiness aspect, though, the CFGs it produces are invariably full of "GOTO"-like structures, making them look rather suspicious—and therefore prone to being attacked.

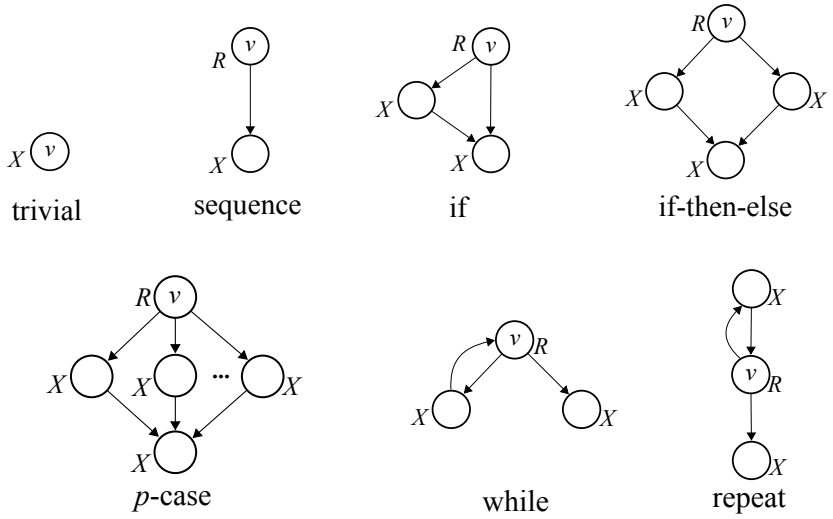

Fig. 1 Statement graphs, the building blocks of Dijkstra graphs, with indications of expansible (X) and regular (R) nodes

The CFGs produced by the watermarking scheme proposed in the next session, on the other hand, remain structured, i.e., they belong to the class of Dijkstra graphs [15]. We recall that the basic constituent blocks of Dijkstra graphs are the following *statement graphs*: (i) the *trivial* graph; (ii) the *sequence* graph; (iii) the *if* graph; (iv) the *if-then-else* graph; (v) the *p-case* graph, for $p \geq 3$; (vi) the *while* graph; and (vii) the *repeat* graph (see Figure 1). Any graph that cannot be obtained by combinations of these building blocks (basically by replacing *expansible* nodes, marked with a 'X' in Figure 1, with some statement graph) is not a Dijkstra graph, and cannot ever be the CFG of a structured program.

3 A structured watermark

In this section we present a graph-based codec for structured programs, one which checks all the boxes in the aforementioned list of desired properties in a watermarking scheme (see Section 2).

- ✓ Data rate: other than those constituting a Hamiltonian path that is so to speak the spinal cord of our watermark, there is a one-to-one correspondence between its (non-path) edges and the bits of the encoded identifier, so its size is linear in the number of encoded bits;
- ✓ Performance: both encoding and decoding algorithms can be implemented to run in linear time in the size of the identifier;
- ✓ Resilience: a removed edge from the proposed watermark may corrupt at most one bit of the binary representation of the identifier, not interrupting the decoding process; hence, distortion attacks flipping b bits may be dealt with after the decoding via

standard mechanisms for error detection/correction [32], provided a suitable number $f(b)$ of redundancy bits has been added to the identifier prior to the encoding; and

✓ Stealthiness: all watermark graphs produced by our encoding algorithm belong to the class of Dijkstra Graphs; moreover, the same identifier may originate *diverse* [3] watermarks along distinct executions of the algorithm, making it harder to spot watermarks by the same author/proprietor via brute force diffing.

3.1 Encoding

Algorithm 3.1 describes the basic steps to encode an n-bit binary identifier B onto a graph G, created initially with $n+2$ vertices labeled from 1 to $n+2$. First, we create a (Hamiltonian) path H with all $n+2$ vertices of G, arranged in ascending order of label along the path. During the execution, the algorithm will certainly add new edges to G and, possibly, a few more vertices.

The edges of the watermark graph may be divided into three groups: *path edges*, those that constitute H, the skeleton of the watermark graph; *back edges*, those that connect a vertex v to some vertex w, with $w < v$; and *forward edges*, edges directed from vertex v precisely to vertex $v+2$.

The two last vertices of H and all destination vertices of forward edges are called *mute* vertices. They only present a structural role, not directly mapping to bits of the encoded identifier, and will never be the origin of back or forward edges. Mute vertices shall be skipped by the decoding algorithm. Every time a vertex is muted by the algorithm by the addition of a forward edge $[v \to v+2]$, a new vertex is included in the graph (at the end of the Hamiltonian path, which increases in size with the addition of a new path edge to the new vertex). This way, the number of non-mute vertices of G will always correspond to the number of bits of the identifier.

The first vertex of H, labeled 1, is not mute. It corresponds to the very first bit in B, which is always a '1' (leading zeroes are dropped), and its out-degree will always be 1 (the path edge $[1 \to 2]$ being its only outgoing edge).

Each bit indexed $i \in \{2, \ldots, n\}$ in B may lead to the addition of a back edge or a forward edge leaving the i-th non-mute vertex along H, in accordance to the rules stated next.

A bit '1' with index $i = 2$ in B may originate either a back edge or a forward edge leaving vertex 2 in H—the choice is random. On the other hand, a bit '0' with index $i = 2$ in B originates no back or forward edges leaving vertex 2.

Along with the outgoing path edge, each vertex $v \geq 3$ in H will be the origin vertex of either a back or a forward edge, except for mute vertices, which have no outgoing forward or back edges. More precisely:

- For each bit '1' with index $i \in \{3, \ldots, n\}$ in B, there will be a back edge going from the i-th non-mute vertex v_i (in the ascending sequence of vertices along H) to a vertex w, such that the difference between the labels of v_i and w is *odd*, or—when there is no *admissible* back edge (definition coming shortly) meeting the required parity of the difference of labels—a forward edge $[v \to v+2]$.

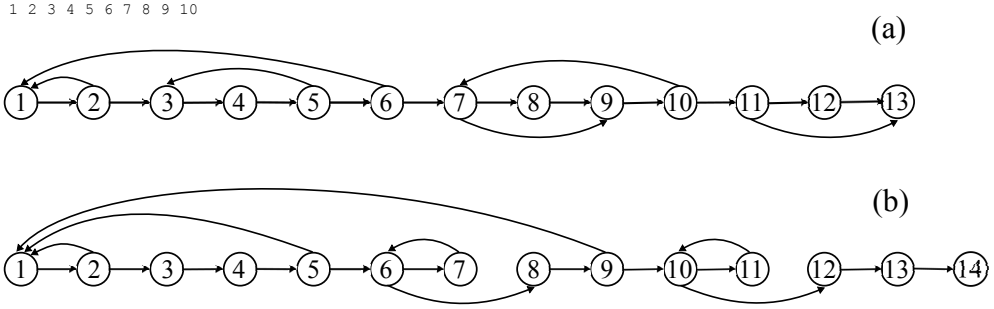

Fig. 2 Identifier $\omega = 795$ encoded as distinct watermark graphs

- For each bit '0' with index $i \in \{3, \ldots, n\}$ in B, there will be a back edge $[w \leftarrow v_i]$ so that $v_i - w$ is *even*, or—when there is no admissible back edge meeting the required parity of the difference of labels—no edge other than the path edge will leave v_i.

A back edge $[w \leftarrow v]$ is considered admissible in G when:

1. $w = v - 1$, if there is in $E(G)$ the forward edge $[v - 1 \rightarrow v + 1]$;
2. There is no back edge $[w \leftarrow v - 1]$ in $E(G)$;
3. There is no back edge $[w' \leftarrow w]$ in $E(G)$, that is, a back edge with origin in w;
4. There is no forward edge $[w - 1 \rightarrow w + 1]$ in $E(G)$;
5. Every back edge $[z' \leftarrow z]$ with $w < z < v$ satisfies $z' > w$.

Any back edge failing to meet the conditions above would lead to a graph which does not belong to the class of Dijkstra graphs, and, therefore, could never correspond to structured code. By construction, the back edges added to G by Algorithm 3.1 are always admissible and may produce, according to the case, basic blocks associated to *while* and *repeat* constructions of Dijkstra graphs (lines 5, 18 and 21).

Note that the addition of a forward edge with origin in the i-th non-mute vertex in H (lines 9 and 26) is immediately followed, in the next iteration, by actions that generate either a *while* or an *if* statement graph, depending on the value of the $(i+1)$-th bit of B (lines 17-20). Note also that the addition of forward edge $[v - 1 \rightarrow v + 1]$ may lead to the removal of the path edge $[v \rightarrow v + 1]$, intentionally breaking apart the original Hamiltonian path while keeping the graph realizability via structured code (line 19).

Figure 2 shows two watermarks generated by Algorithm 3.1 encoding the same decimal identifier $\omega = 795$.

Theorem 1. *Let ω be some positive integer and let n be the size of the binary representation of ω. Algorithm 3.1 encodes the identifier ω in a Dijkstra graph in $O(n)$ time.*

Proof. It is easy to see that the number of forward edges generated by Algorithm 3.1 is, at most, $\lfloor n/3 \rfloor$. Indeed, when the bit of index $i \geq 2$ causes the inclusion of a forward edge (with origin at the i-th non-mute vertex of H), the bit of indices $i+1$ and $i+2$ will never cause the inclusion of forward edges because the back edges $[v_{i+1} - 1 \leftarrow v_{i+1}]$ and $[v_{i+2} - 3 \leftarrow v_{i+2}]$ will be necessarily admissible. Thus, the number of vertices and

Algorithm 3.1: Structured watermark encoding

Input: an identifier ω to be encoded

Output: a Dijkstra graph G (the watermark) encoding ω

let B be the n-bit binary representation of ω, with bit indexes starting at 1
let G be a graph with $V(G) = \{1, \ldots, n+2\}$, $E(G) = \{[v \to v+1], 1 \leq v \leq n+1\}$
if $B[2] =$ '1' **then** choose whether to create a back edge or a forward edge
    if a back edge was chosen **then**
        add back edge $[1 \leftarrow 2]$
    else
        $V(G) = V(G) \cup \{n+3\}$
        add path edge $[n+2 \to n+3]$
        add forward edge $[2 \to 4]$ // muting vertex 4
let $i = 3$ (*current index*) and $v = 3$ (*current vertex*)
while $i \leq n$ **do**
    if forward edge $[v-2 \to v] \in E(G)$ **then**
        $v = v+1$ // v has become a mute vertex, let's skip it
        continue to the next iteration // keeping the current bit index i
    if exists $p > 0$ with the same parity as $B[i]$ s.t. $[v-p \leftarrow v]$ is admissible **then**
        if forward edge $[v-1 \to v+1] \in E(G)$ **then**
            if $B[i] =$ '1' **then**
                add back edge $[v-1 \leftarrow v]$
                remove path edge $[v \to v+1]$
            else do not add forward or back edge with origin in v
        else add admissible back edge $[v-p \leftarrow v]$ chosen uniformly at random
    else
        if $B[i] =$ '1' **then**
            $V(G) = V(G) \cup \{t+1\}$, where $t = |V(G)|$
            add path edge $[t \to t+1]$
            add forward edge $[v \to v+2]$ // muting vertex $v+2$
        else do not add forward or back edge with origin in v
    $i = i+1$; $v = v+1$
return G

edges of the watermark graph satisfies

$$|V(G)| \leq (n+2) + \left\lfloor \frac{n}{3} \right\rfloor = \left\lfloor \frac{4n}{3} \right\rfloor + 2, \tag{1}$$

$$|E(G)| \leq \underbrace{(n+1)}_{\text{initial path}} + \underbrace{\left\lfloor \frac{n}{3} \right\rfloor}_{\text{extra path edges}} + \underbrace{(n-1)}_{\text{back+forward edges}} = \left\lfloor \frac{7n}{3} \right\rfloor. \tag{2}$$

The maximum number of iterations of the main loop of Algorithm 3.1 (line 11) is therefore $O(n)$, since there is one iteration per vertex—the variable v is incremented at each and every iteration.

To show that Algorithm 3.1 runs in $O(n)$ time, it remains to show that lines 15 and 21 are executed in $O(1)$ amortized time, since all the other lines run trivially in constant time (representing G via adjacency lists). In other words, we must make sure that we can randomly pick an admissible back edge with origin at the current vertex v in constant time. That turns out to be a simple task. It suffices to dynamically keep track of all vertices which may be chosen as the destination of some admissible back edge. For any back edge with origin at vertex v, we are interested in vertices $w = v - p$, for some positive integer p, that meet the five admissibility conditions seen in Section 3.1, *and* have the desired parity, i.e., p must be even if and only if the current bit of index i is a '0'.

Note that Condition (1) forces the choice $w = v - 1$ if there is a forward edge $[v - 1 \rightarrow v + 1]$. Note also that Condition (2) requires the removal of vertex w from the set of back edge destination candidates if there is a back edge $[w \leftarrow v - 1]$.

To meet Conditions (3)-(5), we maintain two *lists of candidates*: one containing only even-labeled vertices (initially empty), and another containing only odd-labeled vertices (initially containing only vertex 1). The lists were omitted from the pseudo-code in Algorithm 3.1, for simplicity, and operate as follows. At the end of each iteration, current vertex $v \geq 2$ is pushed to the end of the corresponding list if v has not become the origin of a back edge—therefore meeting Condition (3)—, and there is no forward edge $[v - 1 \rightarrow v + 1]$—meeting Condition (4). As we add a back edge $[w \leftarrow v]$ in G, we remove all vertices $w' > w$ from the candidates lists—enforcing Condition (5).

Note that removing the vertices with label bigger than w from the candidates list that contains w is a trivial task, since the lists are sorted by construction and the choice of w is done by randomly selecting its index in the list (line 21). Now, to remove the vertices $w' > w$ from the list that does not contain w, it suffices to keep track of the size $\rho(w)$ of the *other* list by the time each vertex w was added to its due list. Vertices $w' > w$ belonging to the list that does not contain w will be precisely those at positions greater than $\rho(w)$ in that list. While it is possible that several vertices are removed during a single iteration, each vertex may undergo at most one insertion into (and at most one removal from) a candidates list, completing the proof.

3.2 Decoding

The decoding algorithm comprises four steps. First, we label the watermark vertices in ascending order along the original Hamiltonian path H: $1, 2, \ldots, |V(G)|$. The blocks of the original Hamiltonian path are always consecutive in the CFG, even in the event that said path has been broken up by the encoding algorithm (line 19), hence this labeling can always be achieved trivially. Then, we define the first bit of the binary decoding sequence as '1', which is always true, by construction, since we never codify zeros to the left. We then obtain the bit encoded by the back and forward edges (or the absence thereof) originating at each vertex $v \geq 2$:

- If v is the destination of a forward edge and v is not destination of the path edge $[v-1 \to v]$ (mute vertex), ignore it; otherwise,
- A back edge $[w \leftarrow v]$ such that $v - w$ is odd indicates that the bit encoded by v is a '1';
- A back edge $[w \leftarrow v]$ such that $v - w$ is even indicates that the bit encoded by v is a '0';
- A forward edge $[v \to v+2]$ indicates that the bit encoded by v is a '1';
- And, finally, the absence of back or forward edges originating in v indicates that the bit encoded in v is a '0'.

Finally, we obtain the identifier ω from its binary representation B. Algorithm 3.2 presents the decoding process as pseudo-code.

Algorithm 3.2: Structured watermark decoding

Input: a watermark G

Output: the decimal identifier ω encoded by G

1. Label the vertices of G in ascending order, starting with 1, as they appear in the unique Hamiltonian path of G
2. Create a vector B with a single position filled with '1'
3. Let $i = 2$ (*current index*)
4. **for each** vertex $v \in \{2, |V(G)| - 2\}$ **do**
5. **if** v is non-mute **then**
6. **if** v is origin of a forward edge **then**
7. $B[i] =$ '1'
8. **else if** there is back edge $[w \leftarrow v]$ with $v - w$ odd **then**
9. $B[i] =$ '1'
10. **else**
11. $B[i] =$ '0'
12. $i = i + 1$
13. Let $n = i - 1$ // the index of the last generated bit
14. **return** $\omega = \sum_{i=1}^{n} B[i] \cdot 2^{n-i}$

Theorem 2. *Let ω be an identifier and let G be a Dijkstra graph with N vertices produced by Algorithm 3.1 to encode ω. Algorithm 3.2 correctly extracts ω from G in $\Theta(N)$ time.*

Proof. All steps run clearly in constant time in each of the $|V(G)| - 3$ iterations of the main loop. As for the powers of two in line 14, we employ standard memoization, which demands $O(n)$ multiplications (or left shifts) total, instead of computing each power from scratch for each term of the summation.

4 Encoding example

We now present a complete example of encoding an identifier. Consider again Figure 2, in which $\omega = 795$ is encoded as two distinct, plausible watermarks generated by two independent runs of Algorithm 3.1.

The algorithm starts by obtaining $B = 1100011011$, $n = |B| = 10$, and indexing the bits from 1 to 10 in B, left to right. Next, the algorithm creates a graph with a Hamiltonian path of size $n + 2 = 12$. Note that the watermark shown in Figure 2(a) has $n+3$ vertices, while the watermark shown in Figure 2(b) has $n+4$ vertices, which is of course possible since the encoding process might add new vertices (lines 7 and 24 in Algorithm 3.1).

Vertex 2 corresponds to the bit at position 2 of B, which is a '1'. In this example, Algorithm 3.1 chose to produce the back edge $[1 \leftarrow 2]$ (an odd jump) in both executions (Figures 2(a)-(b)). For vertex 3, which corresponds to the bit '0' in position 3 of B, the algorithm did not have admissible edges because the edge $[1 \leftarrow 3]$ violates Condition (ii) of admissibility. Consequently, the algorithm adds no back or forward edge with origin at 3. The same happens with vertex 4. For vertex 5, corresponding to the bit '0' at position 5 of B, the algorithm had two admissible back edges to choose from, namely $[1 \leftarrow 5]$ and $[3 \leftarrow 5]$. The edge $[3 \leftarrow 5]$ was the one chosen in the execution depicted by Figure 2(a), and $[1 \leftarrow 5]$ was the one corresponding to the watermark in Figure 2(b).

Observe that, up to this point, the algorithm has not muted any vertices in either execution, and the watermarks' vertices have held a one-to-one correspondence with the bits of B so far. For vertex 6, corresponding to the bit '1' at position 6 of B, the first execution of the algorithm added the back edge $[1 \leftarrow 6]$, as can be seen in Figure 2(a); however, in the execution which led to the watermark in Figure 2(b), the algorithm had no admissible back edges to choose from, so it added the forward edge $[6 \rightarrow 8]$, whereupon vertex 8 became "mute" (not corresponding to any bit in B), and an extra vertex was added to the graph.

For vertex 7, which corresponds to the bit '1' in position 7 of B, the algorithm employed the only available choices in each case, namely the forward edge $[7 \rightarrow 9]$ in Fig. 2(a), muting vertex 9 and including a brand new vertex at the end of the original Hamiltonian path, and back edge $[6 \leftarrow 7]$ in Fig. 2(b). Since there is a forward edge $[6 \rightarrow 8]$ in Figure 2(b), the path edge $[7 \rightarrow 8]$ was removed. Notice that vertices $\{6, 7, 8\}$ constitute a *while* block of a Dijkstra graph.

In Figure 2(a), vertex 8 corresponds to the bit '0' at position 8 of B, and no additional edge was added with origin at vertex 8 (because there were no admissible back edges). In Figure 2(b), vertex 8 is a mute vertex (not encoding a bit) and is therefore skipped. Vertex 9, in Figure 2(a), is also mute (and is therefore skipped in the first execution of the algorithm), while in Figure 2(b) it must encode the bit '0' at position 8 of B (note the offset, after the occurrence of the first mute vertex, between vertex labels and bit indexes), and therefore it adds back edge $[1 \leftarrow 9]$.

Vertex 10 corresponds to the bit '1' at position 9 of B in both watermarks. For the watermark in Figure 2(a), the algorithm created back edge $[7 \leftarrow 10]$, chosen among the back edge destination candidates available at that point with the required parity (odd); for the watermark in Figure 2(b), it created the forward edge $[10 \rightarrow 12]$ instead

(given there were no admissible back edges), whereupon vertex 12 became mute and a new vertex was added to the original path.

Vertex 11 corresponds to the bit '1' at position 10 of B in both watermarks. For the watermark in Figure 2(a), the algorithm produced the forward edge $[11 \rightarrow 13]$, since there were no admissible back edges available; for Figure 2(b), it produced the back edge $[10 \leftarrow 11]$, and the path edge $[11 \rightarrow 12]$ was removed. At this point, in both executions, the encoding was completed.

We spare the reader from a step-by-step decoding example, which is all the most straightforward.

5 Codec comparative analysis

In this section, we compare our proposed codec with the codec presented by Chroni et al. [8], given that both codecs are static and graph-based. The comparison is based on desired properties of watermarks: data rate, performance, resilience, and stealthiness. Table 1 summarizes the comparative analysis.

	Data rate		Performance		Resilience	Stealthiness	
	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	Time	Space	Distortive Attacks	Graph Class	Random
Proposed Codec	$4n/3 + 2$	$7n/3$	$O(n)$	$O(n^*)$	Requires error-correction code	Dijkstra Graphs	Yes
Chroni et al. [8]	$2n + 3$	$4n + 3$	$O(n)$	$O(n^*)$	Retrieves $k \leq 2$ edges	Canonical Reducible Permutation Graphs	No

Table 1 Comparison between the Proposed Codec and Chroni et al.'s Codec

In the Table 1, n represents the number of bits in the binary representation B of identifier ω to be encoded in the watermark graph G.

As demonstrated in Section 3, our proposed codec exhibits a smaller number of vertices and edges compared to the codec by Chroni et al.. Furthermore, the asymptotic time and space complexities for both algorithms are linear, with $n^* = 2n + 1$ for Chroni et al.

Regarding resilience, our codec incorporates the use of an error-correction code to recover from distortive attacks. In this process, redundancy bits are inserted into the binary representation B using the chosen error-correction code before it undergoes encoding in the watermark graph G. Subsequently, the error-correction code is employed to restore the original B after the decoding of edges in the watermark graph, as implemented between lines 4-12 of Algorithm 3.2. Therefore, the proposed codec's ability to recover from distortion attacks, causing the removal of edges, is directly influenced by the number of redundancy bits inserted by the employed error-correction code. In contrast, Chroni et al.'s codec consistently recovers from the removal of up to

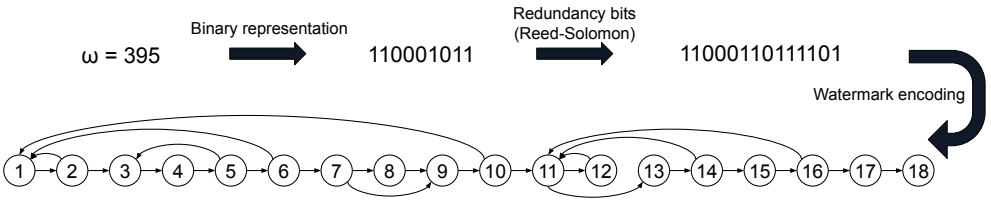

Fig. 3 Example of encoding the identifier $\omega = 795$ in a watermark graph using the Reed-Solomon error correction code to recover distortion attacks by removing up to 2 edges.

2 edges. Experimental results have indicated that removals of 3 to 6 edges are unlikely to lead to incorrect decoding of the identifier for $4 \leq n \leq 10$ [8].

For the preprocessing phase, a viable alternative is to utilize the Reed-Solomon [33] error correction code, renowned for its capability to correct t symbols with the insertion of $2t$ redundancy symbols. Figure 5 illustrates the process of encoding the identifier $\omega = 18$ with 4 bits of redundancy to recover from the removal of up to 2 edges, mirroring the approach employed by Chroni et al.'s codec. It is worth noting that when adopting the Reed-Solomon error correction code and specifying the correction of up to 2 edges, the number of vertices and edges of the watermarks produced by Chroni et al. remains higher than that of the proposed codec. Specifically, in this case, our watermarks have at most $\frac{4n}{3} + 2 + 4 = \frac{4n}{3} + 6$ vertices and $\frac{7n}{3} + 4$ edges.

Regarding resilience, our codec has two important characteristics:

- **Randomization:** Our codec employs randomization to select forward and back edges (or the absence of an edge) for encoding B. This means that a given ω identifier can generate more than one watermark graph. This diversity is essential, as it allows the same identifier to be inserted into different functions of the program through various sequences of instructions (watermark graphs). This complicates the success of removal attacks, as an attacker must identify and eliminate all watermarks inserted in the program. Moreover, using different watermark graphs for the same identifier, corresponding to the identification of the software developer, for instance, in different programs, makes it challenging for attackers to identify watermarks from the same author/owner through half brute force comparison attacks.
- **Absence of "GOTO" Instructions:** The watermark graphs produced by our codec do not require the use of "GOTO" instructions to be encoded in the program. This design choice offers several advantages:
  - The instructions used to encode the watermark in the source code are the same as those employed in the rest of the program, preventing attackers from easily pinpointing these specific parts of the code.
  - The watermark embedding in the program is simplified, as it becomes possible to modify existing sequences of instructions in the code to produce the watermark graph. This results in a reduction in the attack surface, as the code corresponding to the watermark remains executable by the program.
  - Modifying instructions with actual functionality to represent the watermark contributes to the watermark embedding in the code having minimal or no impact on

software quality metrics [34], such as lines of code, cyclomatic complexity, system response time, and code duplication rate.

In summary, our proposed codec demonstrates advancements in terms of resilience to attacks, efficiency, and structure. The elimination of non-structured code elements and the incorporation of randomization contribute to the overall robustness and effectiveness of our watermarking scheme.

6 Conclusion and future works

We have presented a codec that produces watermark graphs belonging to the class of Dijkstra graphs. This feature lends our watermarks a stealthier nature (one less vulnerable to attacks), since the obtained CFGs present no noticeable difference whatsoever with respect to CFGs of real programs, written in real (structured) languages. Our scheme also scores nicely in the diversity aspect, for it employs randomization to produce distinct encodings for the same identifier. Finally, our codec allows for standard error detection and correction techniques, since the non-path edges of the produced watermark graphs bear a one-to-one correspondence to the watermark edges.

The proposed codec (encoding and decoding algorithms) was implemented in Python and successfully tested for all positive integers up to 10^8.[1]

For future directions, we may want to consider defining an upper bound for the maximum in-degree of the watermark vertices. This could mitigate the issue of vertices with small labels ending up being the destination of too many back edges, which corresponds to the CFG of multiple nested loops. Alternatively, it would also be possible to choose the back edge, among the admissible ones, with non-uniform probability, aiming for a more even selection of back edge destinations Moreover, conducting more extensive testing in practical, large-scale environments is essential to thoroughly evaluate the codec's performance, effectiveness, and viability in real-world scenarios.

7 Compliance with Ethical Standards

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

[1] The Software Alliance: Software Management: Security Imperative, Business Opportunity. BSA Global Software Survey, (2018). BSA Global Software Survey. https://www.bsa.org/files/2019-02/2018_BSA_GSS_Report_en_.pdf

[2] Asongu, S.A.: Global software piracy, technology and property rights institutions. Journal of the Knowledge Economy (20/018) (2020) https://doi.org/10.1007/s13132-020-00653-1

[1] The source code can be found in https://www.dropbox.com/s/7elhp32ooj484f8/watermark_RS_012021.py?dl=0.

[3] Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection, 1st edn. Addison-Wesley software security series. Addison-Wesley Professional, USA (2009)

[4] Hamilton, J., Danicic, S.: A survey of static software watermarking. In: 2011 World Congress on Internet Security (WorldCIS-2011), pp. 100–107 (2011). IEEE

[5] Venkatesan, R., Vazirani, V., Sinha, S.: A graph theoretic approach to software watermarking. In: Moskowitz, I.S. (ed.) Information Hiding, pp. 157–168. Springer, Berlin, Heidelberg (2001)

[6] Dey, A., Bhattacharya, S., Chaki, N.: Software watermarking: Progress and challenges. INAE Letters **4**, 65–75 (2019)

[7] Chroni, M., Nikolopoulos, S.D.: An embedding graph-based model for software watermarking. In: 2012 Eighth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, pp. 261–264 (2012). https://doi.org/10.1109/IIH-MSP.2012.69

[8] Chroni, M., Nikolopoulos, S.D., Palios, L.: Encoding watermark numbers as reducible permutation graphs using self-inverting permutations. Discrete Applied Mathematics **250**, 145–164 (2018) https://doi.org/10.1016/j.dam.2018.04.021

[9] Mpanti, A., Nikolopoulos, S.D., Rini, M.: Experimental study of the resilience of a graph-based watermarking system under edge modifications. In: Proceedings of the 21st Pan-Hellenic Conference on Informatics. PCI 2017. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3139367.3139436

[10] Bento, L.M.S., Boccardo, D.R., Machado, R.C.S., Sá, V.G., Szwarcfiter, J.L.: On the resilience of canonical reducible permutation graphs. Discrete Applied Mathematics **234**, 32–46 (2018) https://doi.org/10.1016/j.dam.2016.09.038 . Special Issue on the Ninth International Colloquium on Graphs and Optimization, 2014

[11] Mpanti, A., Nikolopoulos, S.D., Palios, L.: Strong watermark numbers encoded as reducible permutation graphs against edge modification attacks. Journal of Computer Security **31**(2), 107–128 (2023) https://doi.org/10.3233/JCS-210048

[12] Bento, L.M.S., Boccardo, D., Machado, R.C.S., Sá, V.G., Szwarcfiter, J.L.: Towards a provably resilient scheme for graph-based watermarking. In: Brandstädt, A., Jansen, K., Reischuk, R. (eds.) Graph-Theoretic Concepts in Computer Science: 39th International Workshop, WG 2013, Lübeck, Germany, June 19-21, 2013, Revised Papers, pp. 50–63. Springer, New York, NY, USA (2013). https://doi.org/10.1007/978-3-642-45043-3_6

[13] Bento, L.M.S., Boccardo, D.R., Machado, R.C.S., Sá, V.G., Szwarcfiter, J.L.: Full characterization of a class of graphs tailored for software watermarking.

Algorithmica **81**, 2899–2916 (2019) https://doi.org/10.1007/s00453-019-00557-w

[14] Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R. (eds.): Structured Programming. Academic Press Ltd., London, UK, UK (1972)

[15] Bento, L.M.S., Boccardo, D.R., Machado, R.C.S., Miyazawa, F.K., Sá, V.G., Szwarcfiter, J.L.: Dijkstra graphs. Discrete Applied Mathematics **261**, 52–62 (2019) https://doi.org/10.1016/j.dam.2017.07.033 . GO X Meeting, Rigi Kaltbad (CH), July 10–14, 2016

[16] Davidson, R.I., Myhrvold, N.: Method and system for generating and auditing a signature for a computer program. Google Patents. US Patent 5,559,884 (1996)

[17] Arboit, G.: A method for watermarking java programs via opaque predicates. In: In Proc. International Conference on Electronic Commerce Research (ICECR-5) (2002)

[18] Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., Stepp, M.: Dynamic path-based software watermarking. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation. PLDI '04, pp. 107–118. ACM, New York, NY, USA (2004). https://doi.org/10.1145/996841.996856

[19] Cousot, P., Cousot, R.: An abstract interpretation-based framework for software watermarking. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '04, pp. 173–185. ACM, New York, NY, USA (2004). https://doi.org/10.1145/964001.964016

[20] Monden, A., Iida, H., Matsumoto, K., Torii, K., Inoue, K.: A practical method for watermarking java programs. In: 24th International Computer Software and Applications Conference (COMPSAC 2000), 25-28 October 2000, Taipei, Taiwan, pp. 191–197 (2000). https://doi.org/10.1109/CMPSAC.2000.884716

[21] Nagra, J., Thomborson, C.D.: Threading software watermarks. In: Information Hiding, 6th International Workshop, IH 2004, Toronto, Canada, May 23-25, 2004, Revised Selected Papers, pp. 208–223 (2004). https://doi.org/10.1007/978-3-540-30114-1_15

[22] Qu, G., Potkonjak, M.: Analysis of watermarking techniques for graph coloring problem. In: Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design. ICCAD '98, pp. 190–193. ACM, New York, NY, USA (1998). https://doi.org/10.1145/288548.288607

[23] Zeng, Y., Liu, F., Luo, X., Lian, S.: Abstract interpretation-based semantic framework for software birthmark. Comput. Secur. **31**(4), 377–390 (2012) https://doi.org/10.1016/j.cose.2012.03.004

[24] Jacob, R.M., K., P., P.P., A.: Application of visual cryptography scheme in software watermarking. In: 2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184), pp. 1044–1048 (2020)

[25] Mnkash, S.H., Abdulmunem, M.E.: A review of software watermarking. Iraqi Journal of Science **61**(10), 2740–2750 (2020) https://doi.org/10.24996/ijs.2020.61.10.30

[26] Preda, M.D., Ianni, M.: Exploiting number theory for dynamic software watermarking. Journal of Computer Virology and Hacking Techniques (2023) https://doi.org/10.1007/s11416-023-00489-8

[27] Kim, T., Jang, Y., Lee, C., Koo, H., Kim, H.: Smartmark: Software Watermarking Scheme for Smart Contracts, pp. 283–294 (2023). https://doi.org/10.1109/ICSE48619.2023.00035 . Cited by: 0; All Open Access, Green Open Access. https://www.scopus.com/inward/record.uri?eid=2-s2.0-85171746334&doi=10.1109%2fICSE48619.2023.00035&partnerID=40&md5=4d3bc350a272c2ead3c71eef47df42c0

[28] Collberg, C., Thomborson, C.: Software watermarking: Models and dynamic embeddings. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '99, pp. 311–324. ACM, New York, NY, USA (1999). https://doi.org/10.1145/292540.292569

[29] Collberg, C., Kobourov, S., Carter, E., Thomborson, C.: Graph-based approaches to software watermarking. In: Bodlaender, H.L. (ed.) Graph-Theoretic Concepts in Computer Science, pp. 156–167. Springer, Berlin, Heidelberg (2003)

[30] Collberg, C., Kobourov, S., Carter, E., Thomborson, C.: Error-correcting graphs for software watermarking. Lecture Notes in Computer Science **2880**, 156–167 (2003)

[31] Mpanti, A., Nikolopoulos, S.D., Rini, M.: Experimental study of the resilience of a graph-based watermarking system under edge modifications. In: Proceedings of the 21st Pan-Hellenic Conference on Informatics. PCI 2017. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3139367.3139436

[32] Purser, M.: Introduction to Error-correcting Codes. Artech House, ??? (1995)

[33] Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics **8**(2), 300–304 (1960) https://doi.org/10.1137/0108018

[34] Kan, S.H.: Metrics and Models in Software Quality Engineering, 2nd edn. Addison-Wesley Professional, Delhi (2002)