Structured Watermarks for Structured Software

Lucila M. S. Bento [a], Davidson R. Boccardo [b], Raphael C. S. Machado [c], Vinícius G. Pereira de Sá [d,*] and Jayme L. Szwarcfiter [a,d]

[a] *State University of Rio de Janeiro (UERJ), Brazil*
[b] *Clavis Information Security, Brazil*
[c] *Fluminense Federal University (UFF), Brazil*
[d] *Federal University of Rio de Janeiro (UFRJ), Brazil*

Abstract. Software watermarking is the act of embedding an identifier within the software binary. The identifier can be timely retrieved to reveal authorship/ownership, discouraging piracy. Software watermarking schemes based on embedding some encoded identifier into the control flow graph of the program have been suggested in the literature. We propose a novel graph-based watermarking scheme with four principal features: it is randomized; encoding and decoding can both be implemented to run in linear time; it is small in size; and, more importantly, the watermarks it produces conform to structured code. The diversity granted by the ability to encode the same identifier as different subgraphs and the absence of awkward "GOTO"-like substructures provide our watermarks with greater stealthiness, making them more resilient to common forms of attack.

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 recent 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.

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 [3] embeds the watermark graph in the control flow graph (CFG) of the software to be protected. The CFG, which can be obtained from the software binaries 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

* Corresponding author. E-mail: vigusmao@dcc.ufrj.br.

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.

In spite of enjoying a reasonable level of development, graph-based watermark codecs proposed to date have left room for improvement in terms of resilience to attacks [4, 5]. Among the most studied attack models, two require special attention: distortion attacks and subtraction attacks [6]. In the distortion attack model, the adversary attempts to damage the watermark structure by changing the program code to remove some blocks of code or connections among them. In other words, the adversary indirectly attempts to remove CFG vertices and edges from the marked program. In the subtraction attack, the adversary detects the watermark and attempts to remove it altogether.

Some recent graph-based watermark codecs [5, 7] do present some resilience to distortion and subtraction attacks [8, 9], especially if the attacker does not know where in the CFG the watermark is. However, the watermark graphs produced by those codecs, while guaranteed to belong to a subclass of all possible CFGs called *reducible permutation graphs* [10, 11], do not necessarily present a topology that resembles that of CFGs of *structured* programs, since some of their edges demand the addition of artificial jumps in "GOTO" fashion [12]. Such artificial jumps are at the very least unusual or even impossible in many modern languages, and may generate suspicion in a malicious adversary, increasing their chances of identifying the watermarks in the resulting CFGs and perpetrating attacks.

Our contribution. We propose a linear-time codec for graph-based software watermarking, where the code corresponding to the watermark is strictly *structured*, as formally defined in Dijkstra's seminal paper [12]. Thus, when protecting structured software, the resulting, watermarked CFGs will remain in the class of the so-called "Dijkstra graphs" (which correspond precisely to CFGs of structured programs [13]), unlike all previous graph-based watermarks we know of, where (non-structured) "GOTO" statements are inevitably called for. Additionally, our encoding algorithm employs randomization to produce distinct watermarks for the same identifier upon different executions. This behavior makes it less likely that a watermark can be spotted by comparing different programs with watermarks encoding the same identifier (e.g., 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 latter feature contrasts with that of other watermark schemes, where the removal of a small number of edges may damage the entire structure, making it impossible to recover the encoded identifier.

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 5 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 [14] introduced the software watermarking concept for software protection. A lot of research on software watermarking was carried out ever since, and different techniques were employed, including opaque predicates, register allocation, abstract interpretation, dynamic paths, and visual cryptographic scheme [15–22].

Colberg et al. [23] 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. [3] introduced a static graph-based watermark scheme. They encode an integer identifier ω as a graph G. The watermarked program results from changing the original source code, thus also the executable binaries, in such a way that G appears as a subgraph of the (modified) program's CFG.

Some years later, Colberg et al. [24] 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 and Nikolopoulos [7], inspired by the work of Colberg et al. [25], 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* [10], whereupon it was also demonstrated that the watermarking scheme proposed in [7] can withstand attacks in the form of $k \leqslant 2$ edge removals, and there is an infinite number of watermark instances generated by their codec which get irremediably damaged by $k = 3$ edge removals. Mpandi et al. [26] investigated experimentally the resilience to edge modification of that same scheme. The results show that attacks modifying k edges, $3 \leqslant k \leqslant 7$, 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.

The CFGs produced by the watermarking scheme proposed in the next session, on the other hand, remain scructured, i.e., they belong to the class of Dijkstra graphs [13]. 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 $p \geqslant 3$; (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.

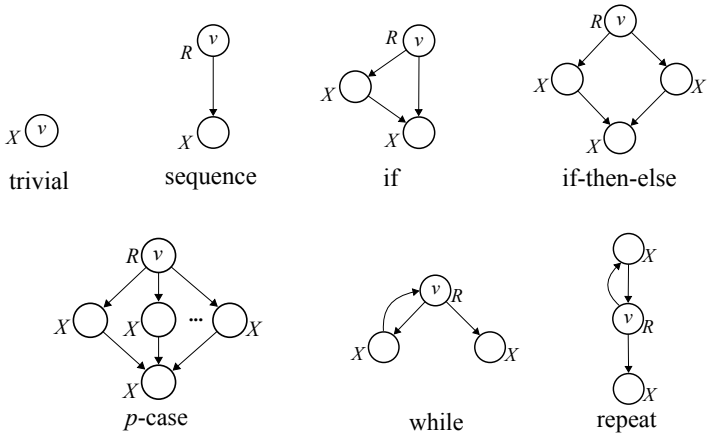

Figure 1. Statement graphs, the building blocks of Dijkstra graphs, with indications of expansible (X) and regular (R) nodes

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, 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* [6] 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 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 $[v \rightarrow v+2]$, 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 $[1 \rightarrow 2]$ 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 $v \geqslant 3$ 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 $[v \rightarrow v+2]$.
- 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 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 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 1 encodes the identifier ω in a Dijkstra graph in $O(n)$ time.*

Algorithm 1: Structured watermark encoding

Input: an identifier ω to be encoded
Output: a Dijkstra graph G (the watermark) encoding ω

(1) **let** B be the n-bit binary representation of ω, with bit indexes starting at 1
(2) **let** G be a graph with $V(G) = \{1, \ldots, n+2\}$, $E(G) = \{[v \to v+1],\ 1 \leqslant v \leqslant n+1\}$
(3) **if** $B[2] =$ '1' **then** choose whether to create a back edge or a forward edge
(4) **if** a back edge was chosen **then**
(5) add back edge $[1 \leftarrow 2]$
(6) **else**
(7) $V(G) = V(G) \cup \{n+3\}$
(8) add path edge $[n+2 \to n+3]$
(9) add forward edge $[2 \to 4]$ // muting vertex 4
(10) **let** $i = 3$ (*current index*) and $v = 3$ (*current vertex*)
(11) **while** $i \leqslant n$ **do**
(12) **if** forward edge $[v-2 \to v] \in E(G)$ **then**
(13) $v = v+1$ // v has become a mute vertex, let's skip it
(14) **continue** to the next iteration // keeping the current bit index i
(15) **if** exists $p > 0$ with the same parity as $B[i]$ s.t. $[v-p \leftarrow v]$ is admissible **then**
(16) **if** forward edge $[v-1 \to v+1] \in E(G)$ **then**
(17) **if** $B[i] =$ '1' **then**
(18) add back edge $[v-1 \leftarrow v]$
(19) remove path edge $[v \to v+1]$
(20) **else** do not add forward or back edge with origin in v
(21) **else** add admissible back edge $[v-p \leftarrow v]$ chosen uniformly at random
(22) **else**
(23) **if** $B[i] =$ '1' **then**
(24) $V(G) = V(G) \cup \{t+1\}$, where $t = |V(G)|$
(25) add path edge $[t \to t+1]$
(26) add forward edge $[v \to v+2]$ // muting vertex $v+2$
(27) **else** do not add forward or back edge with origin in v
(28) $i = i+1$; $v = v+1$
(29) **return** G

Proof. It is easy to see that the number of forward edges generated by Algorithm 1 is, at most, $\lfloor n/3 \rfloor$. Indeed, when the bit of index $i \geqslant 2$ 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 edges of the watermark graph satisfies

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

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

The maximum number of iterations of the main loop of Algorithm 1 (line 11) is therefore $O(n)$, since there is one iteration per vertex—the variable v is incremented at each and every iteration.

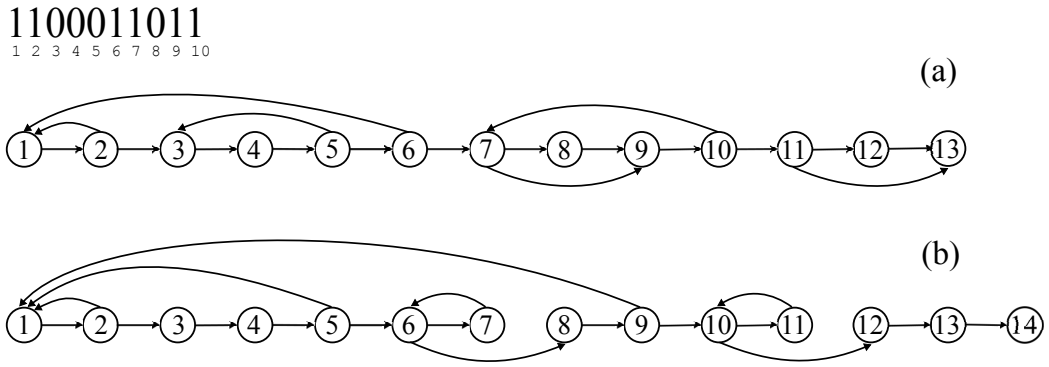

Figure 2. Identifier $\omega = 795$ encoded as distinct watermark graphs

To show that Algorithm 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 pseudocode in Algorithm 1, for simplicity, and operate as follows. At the end of each iteration, current vertex $v \geqslant 2$ 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

Algorithm 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** $\omega = \sum_{i=1}^{n} B[i] \cdot 2^{n-i}$

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 $v \geqslant 2$:

- if v is the destination of a forward edge and v is not destination of the path edge $[v - 1 \rightarrow v]$ (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 $[v \rightarrow v + 2]$ 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 2 presents the decoding process as pseudocode.

Theorem 2. *Let ω be an identifier and let G be a Dijkstra graph with N vertices produced by Algorithm 1 to encode ω. Algorithm 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 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 1).

Vertex 2 corresponds to the bit at position 2 of *B*, which is a '1'. In this example, Algorithm 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. 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 think about defining an upper bound to the maximum in-degree of the watermark vertices. It would mitigate the fact that vertices with a small label might end 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 at a more even selection of back edge destinations.

References

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

[2] S.A. Asongu, Global Software Piracy, Technology and Property Rights Institutions, *Journal of the Knowledge Economy* (2020). doi:10.1007/s13132-020-00653-1.

[3] R. Venkatesan, V. Vazirani and S. Sinha, A Graph Theoretic Approach to Software Watermarking, in: *Information Hiding*, I.S. Moskowitz, ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 157–168.

[4] L.M.S. Bento, D.R. Boccardo, R. Machado, V.G. Pereira de Sá and J.L. Szwarcfiter, A randomized graph-based scheme for software watermarking, in: *XIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSEG'14). SBC, Belo Horizonte, Brazil*, 2014, pp. 30–41.

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

[6] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 2009.

[7] M. Chroni and S.D. Nikolopoulos, An Embedding Graph-based Model for Software Watermarking, in: *2012 Eighth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, 2012, pp. 261–264. doi:10.1109/IIH-MSP.2012.69.

[8] A. Mpanti, S.D. Nikolopoulos and M. Rini, Experimental Study of the Resilience of a Graph-Based Watermarking System under Edge Modifications, 2017.

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

[10] L.M.S. Bento, D. Boccardo, R.C.S. Machado, V.G. Pereira de Sá and J.L. Szwarcfiter, Towards a Provably Resilient Scheme for Graph-Based Watermarking, in: *Graph-Theoretic Concepts in Computer Science: 39th International Workshop, WG 2013, Lübeck, Germany, June 19-21, 2013, Revised Papers*, A. Brandstädt, K. Jansen and R. Reischuk, eds, Springer Berlin Heidelberg, 2013, pp. 50–63.

[11] L.M.S. Bento, D.R. Boccardo, R.C.S. Machado, V.G. Pereira de Sá and J.L. Szwarcfiter, Full Characterization of a Class of Graphs Tailored for Software Watermarking, *Algorithmica* **81** (2019), 2899–2916. doi:10.1007/s00453-019-00557-w.

[12] O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare (eds), *Structured Programming*, 1972.

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

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

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

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

[16] C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn and M. Stepp, Dynamic Path-based Software Watermarking, in: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, 2004.
[17] P. Cousot and R. Cousot, An Abstract Interpretation-based Framework for Software Watermarking, in: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, 2004.
[18] A. Monden, H. Iida, K. Matsumoto, K. Torii and K. Inoue, A Practical Method for Watermarking JAVA Programs, in: *24th International Computer Software and Applications Conference (COMPSAC 2000), 25-28 October 2000, Taipei, Taiwan*, 2000, pp. 191–197. doi:10.1109/CMPSAC.2000.884716.
[19] J. Nagra and C.D. Thomborson, Threading Software Watermarks, in: *Information Hiding, 6th International Workshop, IH 2004, Toronto, Canada, May 23-25, 2004, Revised Selected Papers*, 2004, pp. 208–223. doi:10.1007/978-3-540-30114-1-15.
[20] G. Qu and M. Potkonjak, Analysis of Watermarking Techniques for Graph Coloring Problem, in: *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '98, 1998.
[21] Y. Zeng, F. Liu, X. Luo and S. Lian, Abstract Interpretation-based Semantic Framework for Software Birthmark, *Comput. Secur.* **31**(4) (2012), 377–390. doi:10.1016/j.cose.2012.03.004.
[22] R.M. Jacob, P. K. and A. P.P., Application of Visual Cryptography Scheme in Software Watermarking, in: *2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184)*, 2020, pp. 1044–1048.
[23] C. Collberg and C. Thomborson, Software Watermarking: Models and Dynamic Embeddings, in: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, 1999.
[24] C. Collberg, S. Kobourov, E. Carter and C. Thomborson, Graph-Based Approaches to Software Watermarking, in: *Graph-Theoretic Concepts in Computer Science*, H.L. Bodlaender, ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 156–167.
[25] C. Collberg, S. Kobourov, E. Carter and C. Thomborson, Error-correcting graphs for software watermarking, *Lecture Notes in Computer Science* **2880** (2003), 156–167.
[26] A. Mpanti, S.D. Nikolopoulos and M. Rini, Experimental Study of the Resilience of a Graph-Based Watermarking System under Edge Modifications, 2017.
[27] V. Guruswami, *List Decoding of Error-Correcting Codes: Winning Thesis of the 2002 ACM Doctoral Dissertation Competition (Lecture Notes in Computer Science)*, 2005.
[28] M. Tomlinson, C.J. Tjhai, M.A. Ambroze, M. Ahmed and M. Jibril, *Error-Correction Coding and Decoding: Bounds, Codes, Decoders, Analysis and Applications*, 2018.
[29] E.W. Dijkstra, Go-to statement considered harmful, *Comm. ACM* **11** (1968), 174–186.
[30] M. Chroni and S.D. Nikolopoulos, An Efficient Graph Codec System for Software Watermarking, in: *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, 2012, pp. 595–600. doi:10.1109/COMPSACW.2012.116.
[31] M.S. Hecht and J.D. Ullman, Characterizations of Reducible Flow Graphs, *J. ACM* **21**(3) (1974), 367–375–. doi:10.1145/321832.321835.
[32] R.E. Tarjan, Testing flow graph reducibility, *Journal of Computer and System Sciences* **9**(3) (1974), 355–365. doi:10.1016/S0022-0000(74)80049-8.
[33] T.K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*, 2005.
[34] F.J. MacWilliams and N.J.A. Sloane, *The Theory of Error-Correcting Codes*.
[35] R.L. Davidson and N. Myhrvold, Method and System for Generating and Auditing a Signature for a Computer Program, 1996. http://www.delphion.com/details?pn=US05559884__.
[36] H.B. Mann, *Error correcting codes; proceedings of a symposium. Edited by Henry B. Mann*.
[37] M. Purser, *Introduction to error-correcting codes*, Artech House, 1995.
[38] I.S. Reed and G. Solomon, Polynomial Codes Over Certain Finite Fields, *Journal of the Society for Industrial and Applied Mathematics* **8**(2) (1960), 300–304. doi:10.1137/0108018.