Marca d'água estruturada

Lucila Maria de Souza Bento[5] **, Davidson Rodrigo Boccardo**[5]**,**
Raphael Carlos Santos Machado[*3,4] **, Vinícius Gusmão Pereira de Sá**[1] **e**
Jayme Luiz Szwarcfiter[†1,2]

[1] Universidade Federal do Rio de Janeiro (UFRJ)

[2]Universidade do Estado do Rio de Janeiro (UERJ)

[3]Instituto Nacional de Metrologia, Qualidade e Tecnologia (INMETRO)

[4]Clavis Segurança da Informação

[5]Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET-RJ)

lucila, davidson@clavis.com.br, rcmachado@inmetro.gov.br,
vigusmao@dcc.ufrj.br, jayme@nce.ufrj.br

Abstract. *A digital object watermark corresponds to embedded identification data which can be timely retrieved to reveal authorship/ownership, with aims at discouraging piracy. Software watermarking schemes based on immersing an encoded key into control flow graphs have been suggested in the literature. We propose a novel graph-based watermarking scheme with two distinctive features: first, our encoding algorithm employs randomization; second, and more importantly, our watermarks conform to structured code. The ability to encode the key in distinct forms and the absence of awkward goto-like substructures lend our watermarks greater diversity and stealthiness, making them more resilient to subtraction and distortion attacks. A linear-time implementation is given.*

Resumo. *Uma marca d'água em artefato digital corresponde a uma informação de identificação embarcada naquele objeto de forma oculta, podendo ser usada para comprovar autoria/propriedade, com o objetivo de desencorajar a pirataria. Dentre as técnicas de marca d'água de software apresentadas na literatura, destacam-se os esquemas baseados em grafos, nos quais uma chave secreta é codificada e inserida no grafo de fluxo de controle do programa. Neste artigo, apresentamos um novo esquema de marca d'água baseado em grafos com duas características principais: nosso algoritmo de codificação emprega aleatoriedade; e, o que é mais importante, nossas marcas d'água estão em conformidade com códigos estruturados. A capacidade de codificar uma mesma chave de distintas formas e a ausência de subestruturas semelhantes a peculiares* goto*'s conferem maior diversidade e furtividade às nossas marcas d'água, tornando-as mais resilientes a ataques de subtração e distorção. Apresentamos também uma implementação em tempo linear.*

[*]Autor apoiado por CNPq e FAPERJ.
[†]Autor apoiado pelo CNPq.

1. Introdução

Marcas d'água vêm sendo utilizadas ao longo dos séculos para estabelecer autenticidade, autoria ou propriedade de objetos. No início da década de 1990, esse conceito foi introduzido no contexto de proteção de software, como meio de impedir, ou pelo menos desencorajar, a pirataria de software [Davidson and Myhrvold 1996]. Desde esse momento, muitas pesquisas sobre marca d'água de software foram realizadas, e têm sido empregadas distintas como incluindo predicados opacos, alocação de registradores, interpretação abstrata e caminhos dinâmicos [Arboit 2002, Collberg et al. 2004, Cousot and Cousot 2004, Monden et al. 2000, Nagra and Thomborson 2004, Qu and Potkonjak 1998, Zeng et al. 2012]. A informação exata que é incluída no software pode variar entre identificação de autoria até identificação de propriedade, neste caso para que uma auditoria feita em cópias ilegais do software permita saber de qual cópia autêntica aquelas se originaram. Idealmente, uma marca d'água deve ser de difícil remoção.

Os esquemas de marcas d'água baseados em grafos consistem em algoritmos de codificação/decodificação (*codecs*) que traduzem informações de identificação na forma de um grafo, que em geral pertence a alguma classe especialmente escolhida, e algoritmos de embarcação/extração para inserir/reconhecer marcas d'água existentes em um software. O primeiro esquema de marca d'água para proteção de software logo veio a inspirar a criação de um esquema baseado em grafos, no qual a marca d'água é embarcada no grafo de fluxo de controle (GFC) do software a ser protegido [Venkatesan et al. 2001]. O GFC, que pode ser obtido por ferramentas para análise estática de código, representa todas as sequências de computações possíveis das intruções do programa na forma de um grafo direcionado, cujos vértices são blocos de código estritamente sequenciais e as arestas indicam possíveis relações de precedência entre esses blocos. O algoritmo de embarcação, basicamente, constrói código fictício que será inserido no corpo do programa, a partir de alguma posição secreta predefinida. Este código corresponde exatamente à estrutura da marca d'água pretendida, de modo que o GFC do programa possua um trecho que corresponda exatamente ao grafo que codifica a marca d'água desejada.

Apesar dos algoritmos de embarcação/extração de marcas d'água estarem atualmente num nível razoável de desenvolvimento [Chroni and Nikolopoulos 2012b, Collberg and Nagra 2009], os *codecs* propostos até o momento deixam bastante espaço para melhorias no que se refere à resiliência a ataques. Dentre os modelos de ataque mais estudados, dois deles exigem atenção especial: ataques de distorção e de subtração [Collberg and Thomborson 1999]. No modelo de ataque de distorção, o adversário tenta danificar a estrutura da marca d'água realizando alterações no código do programa, a fim de remover alguns blocos de código ou ligação entre eles, isto é, o adversário tenta indiretamente remover vértices e arestas do GFC do programa marcado. Uma alternativa para mitigar esse tipo de ataque consiste no uso de códigos de correção de erros amplamente conhecidos na literatura [Reed and Solomon 1960, Mann 1968, Purser 1995, Guruswami 2005]. No ataque de subtração, o adversário detecta a presença da marca d'água e tenta removê-la completamente. Para evitar esse tipo de ataque, a marca d'água deve ser tão parecida quanto possível com o código do software protegido, para que o adversário tenha dificuldade em identificá-la após uma ação de engenharia reversa no binário do software.

Recentemente, Chroni e Nikolopoulos apresentaram um engenhoso *codec* [Chroni and Nikolopoulos 2012a] em que a chave é codificada em um grafo pertencente à classe dos grafos de permutação redutíveis [Collberg et al. 2003], possuindo alguma resiliência a ataques [Bento et al. 2013]. Tais grafos são uma subclasse dos GFCs, o que os torna passíveis de serem incorporados ao GFC de um programa. No entanto, sua topologia não corresponde à de um GFC de um *programa estruturado*, uma vez que, para fazer surgir algumas de suas arestas, é frequentemente necessária a adição de saltos (*jumps*) artificiais no programa, isto é, acaba-se tendo que recorrer a instruções pouco usuais (e muito pouco recomendáveis), como o *goto* [Dijkstra 1968]. Como já mencionamos, uma propriedade desejada em uma marca d'água é sua capacidade de estar bem disfarçada no código original, impedindo que seja encontrada e removida. Como as marcas d'água produzidas por este *codec* só podem ser inseridas no código do programa por meio de tais instruções peculiares, estas tenderiam a gerar suspeitas, aumentando as chances de sucesso de um adversário.

Neste artigo, propomos um novo *codec* para marca d'água baseada em grafos. O *codec* proposto utiliza randomização para obter *diversidade*, uma propriedade tida como de grande importância pela comunidade [Collberg and Nagra 2009]. Em resumo, a estrutura das marcas d'água produzidas pelo nosso *codec* é definida a partir de escolhas aleatórias realizadas durante a execução do algoritmo de codificação. Esta característica torna menos provável que uma marca d'água possa ser detectada através de ataques de comparação de força bruta (realizados por ferramentas de *diff*) entre diferentes programas marcados pelo mesmo autor ou para o mesmo proprietário. Uma versão rudimentar desse *codec* for apresentada em [Bento et al. 2014]. O diferencial das marcas d'água apresentadas aqui está no fato de poderem finalmente ser embarcadas adicionando-se código fictício *estruturado* — como definido por Dijkstra [Dahl et al. 1972] — no programa original. Em outras palavras, ao se proteger um programa estruturado, o GFC resultante, com a marca d'água embarcada, continuará pertencendo à classe dos grafos que correspondem a programas estruturados (grafos de Dijkstra), como estudado em [Bento et al. 2015].

Na Seção 2, apresentamos o novo *codec*, ilustrado a seguir por um exemplo completo (codificação/decodificação), na Seção 3. Na Seção 4, descrevemos uma implementação em tempo linear para os algoritmos de codificação e decodificação. A Seção 5 contém nossas considerações finais.

Seja G um grafo. Denotaremos por $V(G)$ e $E(G)$, ao longo do texto, o conjunto de vértices e o conjunto de arestas de G, respectivamente. Para $u, v \in V(G)$, uma aresta de u a v é representada $[u \to v]$, ou, equivalentemente, $[v \leftarrow u]$. Um *caminho* em um grafo G de tamanho k é um sequência de vértices $v_0, v_1, \ldots, v_k$ tal que $[v_{i-1} \to v_i] \in E(G)$ para $i = 1, 2, \ldots, k$. Um caminho v_0, v_1, l, v_k, com $k > 1$, é um *ciclo* de tamanho $k + 1$ se $[v_0 \to v_k] \in E(G)$. Um caminho *hamiltoniano* em um grafo G é um caminho no qual todos os vértices de G aparecem exatamente uma vez.

2. Marca d'água estruturada

Nesta seção, apresentamos nossa proposta de *codec* para marca d'água de software estruturado. Tendo em vista as características desejadas em um esquema de marca d'água (resiliência, furtividade, diversidade, eficiência em tempo e espaço), adiantamos algumas das propriedades principais do *codec* que será a seguir proposto.

- Os grafos produzidos pelo algoritmo de codificação pertencem à classe dos GFCs de programas estruturados.
- Uma aresta removida da marca d'água proposta pode corromper no máximo um bit da representação binária da chave codificada, não interrompendo o processo de decodificação. Assim sendo, ataques de distorção podem ser detectados *depois* da decodificação através de mecanismos usuais para detecção e correção de erros, de forma que se possa detectar e corrigir uma marca d'água em que até b bits, para b arbitrário, tenham sido corrompidos — desde que, evidentemente, um número adequado $f(b)$ de bits de redundância tenha sido acrescentado à chave anteriormente à codificação.
- O algoritmo de codificação executa algumas ações de maneira randomizada, o que significa que uma mesma chave pode originar marcas d'água diferentes (em execuções distintas do algoritmo).
- Os algoritmos de codificação e decodificação podem ser implementados em tempo e espaço lineares no tamanho (quantidade de bits) da chave.

2.1. Codificação

O Algoritmo 1 descreve o passo-a-passo da codificação. Em primeiro lugar, é criado um caminho hamiltoniano H com os $n + 2$ vértices (iniciais) de G, dispostos em ordem crescente de rótulo ao longo do caminho. Durante a execução do algoritmo, novas arestas (e, possivelmente, vértices) serão acrescentadas. As arestas do grafo obtido podem ser divididas em três grupos: *arestas de caminho*, que são as arestas que compõem H; *arestas de retorno*, que são arestas que ligam um vértice v a algum vértice anterior a v em H; e *arestas de avanço*, que são arestas que ligam um vértice v ao vértice $v + 2$.

O primeiro vértice do caminho hamiltoniano H, de rótulo 1, sempre irá corresponder a um bit '1' em B (pois toda chave, em binário, começa com um bit '1'), e seu grau de saída será sempre 1 (a aresta do caminho hamiltoniano é sua única aresta de saída).

Um vértice é considerado *mudo* quando tem função meramente estrutural, não estando mapeado diretamente a um bit da chave, e sendo portanto ignorado pelo algoritmo de decodificação. Os dois últimos vértices de H são vértices mudos, assim como são mudos os vértices que são destinos de arestas de avanço. A cada vez que um vértice $v+2$ é tornado mudo pelo algoritmo (através da inclusão da aresta $[v \rightarrow v + 2]$), um novo vértice é acrescentado ao grafo (e ao caminho hamiltoniano original, com a adição de nova aresta), de forma que o número de vértices não-mudos do grafo permaneça sempre igual 'a quantidade n de bits da chave.

Uma aresta de retorno $[w \leftarrow v]$ é considerada *admissível* em G quando são satisfeitas as cinco condições seguintes:

(i) $w = v - 1$, caso exista em $E(G)$ a aresta de avanço $[v - 1 \rightarrow v + 1]$;
(ii) não existe em $E(G)$ a aresta de retorno $[w \leftarrow v - 1]$;
(iii) não existe em $E(G)$ qualquer aresta de retorno $[w' \leftarrow w]$, isto é, com origem em w;
(iv) não existe em $E(G)$ a aresta de avanço $[w - 1 \rightarrow w + 1]$;
(v) toda aresta de retorno $[z' \leftarrow z]$ com $w < z < v$ (isto é, com origem entre w e v ao longo de H) satisfaz $z' > w$ (isto é, a aresta chega em um vértice posterior a w ao longo de H).

Algoritmo 1: Codificação de marca d'água estruturada

Entrada: uma chave inteira ω a ser codificada
Saída: um grafo de Dijkstra G (marca d'água) codificando ω

1. seja B a representação binária de ω, e seja $n = |B|$
2. crie um grafo G, com conjunto inicial de vértices $V(G) = \{1, \ldots, n+1\}$
   e conjunto inicial de arestas $E(G) = \{[v \to v+1]$, para $1 \leq v \leq n+1\}$
3. seja $i = 2$ (o *índice corrente*), e seja $v = 2$ (o *vértice corrente*)
4. **enquanto** $i \leq n$:
5. **se** existe uma aresta de avanço $[v-2 \to v]$:
6. faça $v = v+1$, de forma a pular o vértice mudo v
7. **continue** para a próxima iteração do laço, mantendo o mesmo valor do índice corrente i
8. **se** existe pelo menos uma aresta de retorno admissível com origem em v:
9. **se** $v-1$ é a origem de uma aresta de avanço $[v-1 \to v+1]$:
10. **se** $B[i] =$ '1':
11. adicione a aresta de retorno $[v-1 \leftarrow v]$
12. remova a aresta de caminho $[v \to v+1]$
13. // os vértices $v-1, v, v+1$ corresponderão a // um bloco *while* no grafo de Dijkstra G
14. **senão**:
15. não adicione aresta de retorno ou avanço com origem em v
16. // os vértices $v-1, v, v+1$ corresponderão a // um bloco *if*...*then* no grafo de Dijkstra G
17. **senão**:
18. adicione uma aresta $[v-k \leftarrow v]$ escolhida aleatória e uniformemente dentre as arestas de retorno admissíveis com origem em v
19. // os vértices $v-k, v-k+1, \ldots, v$ corresponderão a // um bloco *repeat* no grafo de Dijkstra G
20. **senão**:
21. **se** $B[i] =$ '1':
22. faça $V(G) = V(G) \cup \{t+1\}$, onde t é o rótulo do maior vértice atualmente em $V(G)$
23. acrescente a aresta de caminho $[t \to t+1]$
24. acrescente a aresta de avanço $[v \to v+2]$
25. // o vértice $v+2$ será um vértice mudo
26. faça $i = i+1$, e faça $v = v+1$
27. **retorne** G

Com exceção dos vértices mudos, que são origens apenas de arestas de caminho, cada vértice $v > 1$ poderá ser o vértice de origem de até uma aresta de retorno ou de avanço. Mais precisamente:

- para cada bit '1' de índice $i \in \{2, \ldots, n\}$ na representação binária B da chave, teremos uma aresta de retorno indo do i-ésimo vértice não-mudo v_i (na sequência

dada por H) até um vértice w tal que a diferença entre os rótulos de v_i e w seja *ímpar*, ou — quando não houver aresta de retorno admissível satisfazendo a paridade desejada da diferença dos rótulos — uma aresta de avanço $[v \rightarrow v+2]$;
- para cada bit '0' de índice $i \in \{2, \ldots, n\}$ em B, teremos analogamente uma aresta de retorno $[w \leftarrow v_i]$ tal que $v_i - w$ é *par*, ou — quando não houver aresta de retorno admissível satisfazendo a paridade desejada da diferença dos rótulos — não acrescentaremos aresta com origem no i-ésimo vértice não-mudo v_i.

Uma aresta de retorno que não satisfizesse as condições acima daria origem, invariavelmente, a um grafo não pertencente à classe dos grafos de Dijkstra, e que, portanto, não poderia aparecer como GFC de código estruturado. Por construção, as arestas de retorno adicionadas a G pelo Algoritmo 1 são sempre admissíveis, e produzem, de acordo com o caso, blocos básicos associados aos construtos *while* e *repeat* dos grafos de Dijkstra (linhas 9–12 e 18–19).

Note que a adição de uma aresta de avanço $[v_i \rightarrow v_i + 2]$ com origem no i-ésimo vértice não-mudo do caminho hamiltoniano, em qualquer das iterações do Algoritmo 1, é sempre seguida, já na iteração seguinte, de ações que levam à geração de um bloco *while* ou de um bloco *if*...*then*, que são *statement graphs* constituintes dos grafos de Dijkstra [Bento et al. 2015]. O bloco exato a ser gerado dependerá do valor do $(i+1)$-ésimo bit de B (linhas 9–12 e 15–16). Pela mesma razão, note também que a adição da aresta de avanço $[v \rightarrow v+2]$ implica sempre a remoção da aresta de caminho $[v \rightarrow v+1]$, rompendo o caminho hamiltoniano original.

2.2. Decodificação

O algoritmo de decodificação consiste de quatro etapas. Primeiro, os vértices da marca d'água são rotulados ao longo do caminho hamiltoniano original em ordem crescente: $1, 2, \ldots, |V(G)|$. Tal rotulação sempre pode ser realizada, já que os blocos do caminho hamiltoniano original (ainda que este tenha sido rompido ao longo da codificação) são consecutivos no GFC. Em seguida, definimos o primeiro bit do binário como '1' (o que é sempre verdade, por construção, uma vez que jamais codificamos zeros à esquerda). Passamos então a obter as informações codificadas pelas arestas de retorno e avanço (ou por sua ausência) com origem em cada um dos vértices de rótulo $v \geq 2$:

- se v é destino de uma aresta de avanço (vértices mudo), ignore-o; senão,
- uma aresta de retorno $[w \leftarrow v]$ tal que $v - w$ é ímpar (respectivamente, par) indica que o bit codificado em v é '1' (respectivamente, '0');
- uma aresta de avanço $[v \leftarrow v+2]$ indica que o bit codificado por v é '1';
- e, finalmente, a não existência de aresta de retorno ou avanço com origem em v indica que o bit codificado em v é '0'.

Ao final, o algoritmo percorre os bits de B calculando o valor inteiro da chave ω da forma usual. O Algoritmo 2 apresenta em pseudo-código o processo de decodificação.

3. Um exemplo completo

A Figura 1 ilustra duas marcas d'água geradas pelo Algoritmo 1 para a chave $\omega = 397$, cuja representação binária, determinada pelo passo 1, é $B = 110001101$, com $n = 9$. A marca d'água apresentada na Figura 1(a) possui $n+3$ vértices, enquanto a marca d'água da Figura 1(b) possui $n+4$ vértices. O vértice 2 corresponde ao bit da posição 2 em B,

Algoritmo 2: Decodificação de marca d'água estruturada

Entrada: uma marca d'água G
Saída: a chave ω codificada por G

1. rotule os vértices de G em ordem ascendente, partir de 1, ao longo do único caminho hamiltoniano de G
2. crie um vetor B, inicialmente com uma única posição, preenchida com '1'
3. seja $i = 2$ (o *índice corrente*)
4. **para cada** vértice $v \in \{2, |V(G)| - 2\}$:
5. **se** v não for mudo (destino de uma aresta de avanço):
6. **se** v é origem de uma aresta de avanço:
7. $B[i] =$ '1'
8. **senão se** existe aresta de retorno $[w \leftarrow v]$ com $v - w$ ímpar:
9. $B[i] =$ '1'
10. **senão**:
11. $B[i] =$ '0'
12. faça $i = i + 1$
13. seja $n = i - 1$ (o índice do último bit gerado)
14. **retorne** $\omega = \sum_{i=1}^{n} B[i] \cdot 2^{n-i}$

que é um '1', e assim o algoritmo produz a aresta de retorno $[1 \leftarrow 2]$ (um salto ímpar) nas duas execuções. Para o vértice 3, que corresponde ao bit '0' na posição 3 de B, o algoritmo não dispunha de arestas admissíveis (a aresta $[1 \leftarrow 3]$, embora apresente o desejado salto par, viola a segunda condição de admissibilidade). Dessa forma, nenhuma aresta adicional a partir de 3 é acrescentada. O mesmo acontece com o vértice 4. Para o vértice 5, correspondente ao bit '0' na posição 5 de B, o algoritmo dispunha de duas arestas de retorno admissíveis: $[1 \leftarrow 5]$ e $[3 \leftarrow 5]$. A aresta de $[3 \leftarrow 5]$ foi a escolhida na execução da Figura 1(a), e $[1 \leftarrow 5]$ na da Figura 1(b).

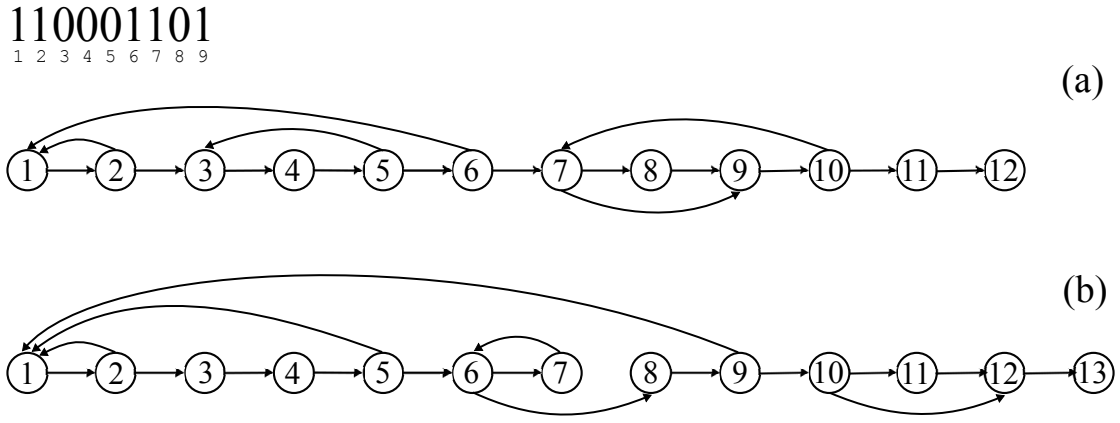

Figura 1. Marcas d'água distintas codificando a chave $\omega = 397$

Note que, até este ponto, o algoritmo não acrescentou nenhuma aresta de avanço, mantendo os rótulos dos vértices equivalentes aos índices dos bits de B. Para o vértice 6, correspondente ao bit '1' na posição 6 de B, para a marca d'água da Figura 1(a) o

algoritmo acrescentou a aresta de retorno $[1 \leftarrow 6]$; no entanto, na execução que levou à marca d'água da Figura 1(b) o algoritmo não dispunha de arestas de retorno admissíveis, e foi portanto levado à adicionar a aresta de avanço $[6 \rightarrow 8]$. Dessa forma, o vértice 8 tornou-se "mudo", implicando a inclusão de um vértice extra e desemparelhando os rótulos dos vértices da marca d'água da Figura 1(b) e os índices dos bits de B.

Dando sequência ao exemplo de codificação, para o vértice 7, que corresponde ao bit '1' na posição 7 de B, o algoritmo emprega as únicas opções disponíveis em cada caso, que são a aresta de avanço $[7 \rightarrow 9]$ na primeira marca d'água, deixando o vértice 9 "mudo" e causando a inclusão de novo vértice, e a aresta de retorno $[6 \leftarrow 7]$, na segunda.

Perceba que, na Figura 1(b), para que os vértices $\{6, 7, 8\}$ correspondessem a um bloco *while* dos grafos de Dijkstra, a aresta de caminho $[7 \rightarrow 8]$ precisou ser removida.

Na Figura 1(a), o vértice 8 representa o bit '0' da posição 8 de B e, portanto, nenhuma aresta adicional, uma vez que não havia arestas de retorno admissíveis. Por outro lado, na Figura 1(b), o vértice 8 é vertice mudo, não codificando bit. Já o vértice 9, na Figura 1(a), é mudo, enquanto na Figura 1(b) está associado à codificação do bit '0' na posição 8 de B. Para tal, o algoritmo adicionou a aresta de retorno $[1 \leftarrow 9]$.

Dando sequência, o vértice 10 corresponde ao bit '1' na posição 9 de B tanto na Figura 1(a) quanto na Figura 1(b). Para a marca d'água da Figura 1(a), o algoritmo escolheu criar a aresta de retorno $[7 \leftarrow 10]$, dentre as diversas opções admissíveis para a paridade desejada do salto (ímpar); para a marca d'água da Figura 1(b), criou a aresta de avanço $[10 \rightarrow 12]$, dado que não havia arestas de retorno admissíveis.

A decodificação das duas marcas d'água é facilmente verificável.

4. Análise

É fácil ver que a quantidade de arestas de avanço adicionadas ao longo do algoritmo é, no máximo, $\lfloor n/3 \rfloor$. De fato, quando o bit de índice $i \geq 3$ ocasiona a inclusão de uma aresta de avanço (com origem no i-ésimo vértice não-mudo de H), os bits de índices $i+1$ e $i+2$ jamais ocasionarão a inclusão de arestas de avanço, pois as arestas de retorno $[v_{i+1} - 1 \leftarrow v_{i+1}]$ e $[v_{i+2} - 3 \leftarrow v_{i+2}]$ serão necessariamente admissíveis. Dessa forma, a quantidade de vértices do grafo marca d'água satisfaz

$$|V(G)| \leq (n+2) + \lfloor n/3 \rfloor = \lfloor 4n/3 \rfloor + 2.$$

O número máximo de iterações do laço principal do Algoritmo 1 é, portanto, $O(n)$, uma vez que temos uma iteração por vértice — a variável v é incrementada a cada iteração.

Para mostrarmos que o Algoritmo 1 pode ser implementado para rodar em tempo $O(n)$, é suficiente garantir que suas linhas 8 e 18 são executadas em tempo $O(1)$, pois todas as demais linhas rodam em tempo constante de forma trivial (representando o grafo G por meio de usuais listas de adjacências). Ou seja, precisamos garantir que se possa escolher em tempo constante uma aresta de retorno admissível com origem no vértice corrente v. Para isso, manteremos um registro dos vértices que são a cada momento *candidatos* a ser o destino de uma aresta de retorno (admissível) com origem no vértice corrente v. Ora, tal conjunto de candidatos é constituído por todo vértice $w < v$ que satisfaça as cinco condiçoes de admissibilidade vistas na Seção 2.1, e além disso possua

a paridade desejada, isto é, a mesma paridade de v, caso o bit corrente, de índice i, seja um '0', ou paridade distinta da de v, caso o bit corrente seja um '1'.

A condição (i) pode ser verificada trivialmente antes mesmo do sorteio do destino admissível w, forçando-se a escolha $w = v-1$ se existir a aresta de avanço $[v-1 \rightarrow v+1]$. O mesmo é feito com relação à condição (ii), removendo-se o vértice w do conjunto de candidatos caso exista a aresta de retorno $[w \leftarrow v-1]$.

Para atender às demais condições, o que fazemos é manter duas pilhas de candidatos, uma que conterá apenas vértices de rótulo par (inicialmente vazia), e outra que conterá apenas vértices de rótulo ímpar (inicialmente contendo apenas o vértice 1). Fazemos também com que, ao final de cada iteração, o vértice corrente $v \geq 2$ seja acrescentado à pilha correspondente caso v não tenha se tornado a origem de uma aresta de retorno, satisfazendo assim a condição (iii), e não haja aresta de avanço $[v-1 \rightarrow v+1]$, respeitando-se assim também 'a condição (iv). Até aqui, sempre em tempo constante. Finalmente, ao acrescentarmos uma aresta de retorno $[w' \leftarrow v]$, removemos da pilha das pilhas de candidatos todo vértice w tal que $w > w'$, garantindo assim que a condição (v) seja atendida. Note que remover os vértices de rótulo maior do w' da pilha que contém o próprio w' é uma tarefa trivial, uma vez que o índice de w' é conhecido. De fato, tal índice acabou de ser escolhido aleatoriamente dentre os índices da pilha correspondente à paridade desejada, apontando w'. Como os candidatos estão empilhados, por construção, em ordem crescente de rótulo, basta apontarmos o fim da pilha para o índice de w', e a remoção está consumada em tempo $O(1)$. Já a remoção em tempo constante dos vértices $w > w'$ da *outra* pilha requer o uso de uma estrutura auxiliar que registra o tamanho de cada pilha no momento em que cada vértice é acrescentado a uma das pilhas[1], como recomendado em [Bento et al. 2014]. Com esse mapa auxiliar (implementado facilmente sobre um vetor com endereçamento direto, identificando-se rótulo de vértice com índice no vetor), podemos recuperar o tamanho que tinha a pilha que não contém w' no momento em que w' foi inserido em sua pilha, isto é, o maior índice contendo um vértice de rótulo menor do que w'. De posse desse índice, basta apontarmos o final da pilha para ele, e teremos assim removido — por meio de uma única operação de tempo constante — todos os vértices com rótulo maior do w'.

Para sortearmos o vértice w dentre os candidatos a destino de um aresta de retorno com origem no vértice corrente v, basta sortearmos um número inteiro aleatório e uniforme entre 1 e o tamanho da pilha de candidatos com a paridade desejada (definida pelo valor do bit corrente) — o que é feito em tempo constante.

O Algoritmo 2 roda claramente em tempo linear na quantidade de vértices da marca d'água, portanto $\Theta(n)$.

5. Conclusão

Apresentamos um *codec* que produz marcas d'água que podem aparecer como subgrafos do GFC por meio apenas da adição de código fictício estruturado. Essa propriedade aumenta a furtividade das marcas d'água produzidas, uma vez que o GFC obtido passa se assemelhar a um GFC típico, correspondente a programa estruturado.

[1]Na verdade, é suficiente registrar o tamanho da pilha que *não está recebendo* v, no momento em que v é acrescentado a uma das pilhas — uma vez que a remoção dos vértices da própria pilha que contem v será feita facilmente, como explicado.

Além disso, diferentemente do que foi mostrado para o esquema de marca d'água de Chroni e Nikolopoulos (capaz de se recuperar de ataques de distorção que causem a remoção de, no máximo, 2 arestas [Bento et al. 2016]), o *codec* proposto pode se recuperar de ataques de distorção que produzam um número arbitrário e predefinido b de arestas removidas, por possibilitar o uso de códigos de correção de erro bit-a-bit.

É também digno de nota que o *codec* proposto, ao empregar randomização, ganha pontos no quesito diversidade, pois apresenta diversas codificações possíveis para uma mesma chave.

Os algoritmos de codificação e decodificação propostos foram implementados em linguagem Python 3, e testados com sucesso para todas as chaves inteiras entre 1 e 10^7. Após a codificação da chave, cada marca d'água obtida foi submetida ao algoritmo de decodificação, e a chave decodificada comparada com a chave original. A Figura 2 ilustra um exemplo de execução.

```
========================================

Options:
  an integer positive key --> for encoding/decoding in verbose mode;
  min_key, max_key (comma-separated) --> for testing over a range;
  <enter> --> quit.

Please type the key: 12345
B = 11000000111001 (n = 14)

Watermark:

       1<<            3<<       1<<       8<<        10<<         1<<        14<<
    1--->2--->3--->4--->5--->6--->7--->8--->9--->10--->11    12--->13--->14--->15--->16--->17
                                              >>12
Testing decoding procedure...
Retrieved key: 12345
1 key(s) encoded/decoded successfully.

========================================
```

Figura 2. Exemplo de saída obtida durante os testes do *codec*

O código-fonte completo (em linguagem Python 3) pode ser encontrado em: `https://www.dropbox.com/s/iqv6o5asfl13wlq/watermark.py`.

Um refinamento possível, que em nada altera as características ou a complexidade do *codec*, é o estabelecimento de um grau de entrada máximo para os vértices. Isso tende a evitar um acúmulo de arestas de retorno com destino nos vértices de rótulo mais baixo (mais à esquerda, no caminho hamiltoniano original), e que participam portanto, como candidatos, de um número maior de sorteios. Isso evitaria, por conseguinte, o aparecimento de uma quantidade excessiva de laços aninhados (com início no mesmo vértice do GFC). Alternativamente, seria também possível escolher a aresta de retorno, dentre as admissíveis, com probabilidade não uniforme, favorecendo os vértices de maior rótulo. A primeira dessas soluções foi de fato implementada em nosso código, onde o grau de entrado máximo do grafo é escolhido aleatória e uniformemente entre 1 e $\sqrt{n}$, adicionando dessa forma um grau extra de aleatoriedade.

Referências

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

Bento, L. M., Boccardo, D. R., Machado, R. C., de Sá, V. G. P., and Szwarcfiter, J. L. (2016). On the resilience of canonical reducible permutation graphs. *Discrete Applied Mathematics (aceito para publicação)*.

Bento, L. M. S., Boccardo, D., Machado, R. C. S., Pereira de Sá, V. G., and Szwarcfiter, J. L. (2013). Towards a provably resilient scheme for graph-based watermarking. In Brandstädt, A., Jansen, K., and Reischuk, R., editors, *Graph-Theoretic Concepts in Computer Science: 39th International Workshop, WG 2013, Lübeck, Germany, June 19-21, 2013, Revised Papers*. Springer Berlin Heidelberg.

Bento, L. M. S., Boccardo, D. R., Machado, R., de Sá, V. G. P., and Szwarcfiter, J. L. (2014). A randomized graph-based scheme for software watermarking. In *XIV Simpósio Brasileiro de Seguraça Informação e de Sistemas Computacionais (SBSEG'14). SBC, Belo Horizonte, Brazil*, pages 30–41.

Bento, L. M. S., Boccardo, D. R., Machado, R., de Sá, V. G. P., and Szwarcfiter, J. L. (2015). The graphs of structured programming. In *13th Cologne Twente Workshop on Graphs and Combinatorial Optimization, Istanbul, Turkey, May 26-28, 2015.*, pages 105–108.

Chroni, M. and Nikolopoulos, S. D. (2012a). An efficient graph codec system for software watermarking. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 595–600.

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

Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., and Stepp, M. (2004). Dynamic path-based software watermarking. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 107–118, New York, NY, USA. ACM.

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

Collberg, C. and Nagra, J. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition.

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

Cousot, P. and Cousot, R. (2004). An abstract interpretation-based framework for software watermarking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 173–185, New York, NY, USA. ACM.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., editors (1972). *Structured Programming*. Academic Press Ltd., London, UK, UK.

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

Dijkstra, E. W. (1968). Go-to statement considered harmful. *Comm. ACM*, 11:174–186.

Guruswami, V. (2005). *List Decoding of Error-Correcting Codes: Winning Thesis of the 2002 ACM Doctoral Dissertation Competition (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Mann, H. B. (1968). *Error correcting codes; proceedings of a symposium. Edited by Henry B. Mann*. Wiley New York.

Monden, A., Iida, H., Matsumoto, K., Torii, K., and Inoue, K. (2000). A practical method for watermarking java programs. In *24th International Computer Software and Applications Conference (COMPSAC 2000), 25-28 October 2000, Taipei, Taiwan*, pages 191–197.

Nagra, J. and Thomborson, C. D. (2004). Threading software watermarks. In *Information Hiding, 6th International Workshop, IH 2004, Toronto, Canada, May 23-25, 2004, Revised Selected Papers*, pages 208–223.

Purser, M. (1995). *Introduction to error-correcting codes*. Artech House.

Qu, G. and Potkonjak, M. (1998). Analysis of watermarking techniques for graph coloring problem. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '98, pages 190–193, New York, NY, USA. ACM.

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., and Sinha, S. (2001). *A Graph Theoretic Approach to Software Watermarking*, pages 157–168. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zeng, Y., Liu, F., Luo, X., and Lian, S. (2012). Abstract interpretation-based semantic framework for software birthmark. *Comput. Secur.*, 31(4):377–390.