PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Universidade Federal do Rio de Janeiro

Lucila Maria de Souza Bento

MARCAS D'ÁGUA BASEADAS EM GRAFOS PARA A PROTEÇÃO DE SOFTWARE

TESE DE DOUTORADO

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LUCILA MARIA DE SOUZA BENTO

MARCAS D'ÁGUA BASEADAS EM GRAFOS PARA A PROTEÇÃO DE SOFTWARE

Rio de Janeiro
2015

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
INSTITUTO TÉRCIO PACITTI DE APLICAÇÕES E PESQUISAS COMPUTACIONAIS
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

LUCILA MARIA DE SOUZA BENTO

MARCAS D'ÁGUA BASEADAS EM GRAFOS PARA A PROTEÇÃO DE SOFTWARE

Tese de Doutorado apresentada ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Doutora em Informática.

Orientadores: Jayme Luiz Szwarcfiter
Vinícius Gusmão Pereira de Sá

Rio de Janeiro
2015

CIP - Catalogação na Publicação

B478m

Bento, Lucila Maria de Souza
Marcas d'água baseadas em grafos para a proteção de software / Lucila Maria de Souza Bento. -- Rio de Janeiro, 2015.
110 f.

Orientador: Jayme Luiz Szwarcfiter.
Coorientador: Vinícius Gusmão Pereira de Sá.
Tese (doutorado) - Universidade Federal do Rio de Janeiro, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Programa de Pós Graduação em informática, 2015.

1. Marca d'água de software. 2. Marca d'água baseada em grafos. 3. Proteção de software. 4. Algoritmos. I. Szwarcfiter, Jayme Luiz, orient. II. Sá, Vinícius Gusmão Pereira de, coorient. III. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a).

Lucila Maria de Souza Bento

Marcas d'água baseadas em grafos para a proteção de software

Tese de Doutorado apresentada ao Departamento de Ciência da Computação do Instituto de Matemática, e Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Doutora em Informática.

Examinada por:

Prof. Dr. Jayme Luiz Szwarcfiter
Orientador

Prof. Dr. Vinícius Gusmão Pereira de Sá
Orientador

Profa. Dra. Lilian Markenzon
Universidade Federal do Rio de Janeiro

Prof. Dr. Fábio Protti
Universidade Federal Fluminense

Prof. Dr. Paulo Eustáquio Duarte Pinto
Universidade do Estado do Rio de Janeiro

Rio de Janeiro
14 de outubro de 2015

Dedico este trabalho a minha família e meus amigos.

AGRADECIMENTOS

Primeiramente, agradeço a Deus por ter me dado essa oportunidade de crescimento pessoal, pelos desafios e pelas pessoas, que de alguma forma, estão presentes no meu cotidiano.

Agradeço a minha família, por me aturar falando coisas que ninguém entende e, mesmo assim, mostrarem interesse. Em especial ao meus pais, pelo exemplo de perseverança e apoio.

Agradeço aos professores Jayme e Vinícius pela paciência e orientação nos últimos anos. Tenho certeza de que sem eles este trabalho não seria possível.

Agradeço também à Raphael Machado e Davidson Boccardo pela participação no desenvolvimento deste trabalho.

Aos colegas e amigos do LabAC com os quais tive o prazer e o privilégio de conviver durante o mestrado e o doutorado.

Aos professores e funcionários do Programa de Pós-Graduação em Informática da Universidade Federal do Rio de Janeiro que de várias formas contribuíram para a conclusão deste trabalho.

RESUMO

BENTO, Lucila Maria de Souza. **Marcas d'água baseadas em grafos para a proteção de software**. 2015. 110 f. Tese (Doutorado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

Com o crescimento da internet, a cópia de documentos digitais tornou-se fácil e acessível, sendo normalmente realizada de modo indiscriminado. Um reflexo desse problema é a pirataria de software. Estima-se que a taxa global de softwares instalados sem licença adequada ultrapassa 43%.

Neste cenário, a técnica de marca d'água surge como uma possível defesa contra a pirataria. Na técnica de marca d'água de software, uma informação de identificação é inserida no software como prova de sua propriedade. Apesar da técnica não impedir a realização de cópias ilegais do software, a marca d'água possibilita o reconhecimento da origem do software distribuído ilegalmente, e, portanto, pode desencorajar a pirataria de maneira significativa.

Esta tese apresenta um estudo detalhado de um algoritmo de marca d'água existente na literatura, concentrando-se principalmente na capacidade da marca d'água ser recuperada após a realização de ataques. Em seguida, é apresentado um algoritmo de marca d'água baseado em grafos que provê uma maior resistência a ataques e outras características importantes discutidas na literatura, como diversidade, eficiência e aumento insignificante no tamanho do programa marcado.

Palavras-chave: Marca d'água de software, marca d'água baseada em grafos, proteção de software, algoritmos.

ABSTRACT

BENTO, Lucila Maria de Souza. **Marcas d'água baseadas em grafos para a proteção de software**. 2015. 110 f. Tese (Doutorado em Informática) - PPGI, Instituto de Matemática, Instituto Tércio Pacitti de Aplicações e Pesquisas Computacionais, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2015.

With the development of the internet, copying digital documents has become easy and affordable, and is usually accomplished indiscriminately. A consequence of this problem is software piracy. It is estimated that the overall rate of installed software without proper licensing exceeds 43%.

In this scenario, one possible defense against piracy is the use of watermarking techniques. Software watermarking tries to insert a identification into the software program as evidence of ownership. Although this will not prevent all illegal copying of software, watermarks make it possible to determine the source of software distributed illegally, and hence would likely discourage piracy to some extent.

In this thesis, we present a detailed study about a software watermarking algorithm existing in the literature. This study focuses on the ability of the watermark to be restored after having been exposed to an attack. Next, we present a graph-based software watermarking algorithm which provides greater resistance to attacks, and other important characteristics discussed in the literature such as diversity, efficiency and insignificant increase in the size of the marked program.

Keywords: software watermarking, graph-based watermarking, software protection, algorithms.

LISTA DE FIGURAS

Figura 1.1: Construção do CFG 15

Figura 3.1: Marca d'água para a chave $\omega = 37$. 30
Figura 3.2: Grafo de fluxo redutível 37
Figura 3.3: Grafo de fluxo redutível auto-rotulável 37
Figura 3.4: Árvores representativas 39
Figura 3.5: (a) Árvore representativa Tipo-1. (b) Árvore representativa Tipo-2. 44
Figura 3.6: Cenários possíveis para o caminho hamiltoniano de uma marca d'água danificada G' após a remoção de até duas arestas. 58
Figura 3.7: Condições (i), (ii) e (iii) do Teorema 6. 67

Figura 4.1: Marcas d'água distintas codificando a chave $\omega = 395$ 87
Figura 4.2: Codificação de $\omega = 395$ com correção de 1 bit 94
Figura 4.3: Decodificação da marca d'água com sucesso 95
Figura 4.4: Decodificação da marca d'água com erro 96

LISTA DE TABELAS

Tabela 3.1: Tempos médios (e desvios-padrão) de decodificação. 78
Tabela 3.2: Frequência relativa $r(n, k)$ de marcas d'água referentes a chaves de tamanho n e k-sinônimas de alguma outra marca d'água, e probabilidade $p(n, k)$ de uma marca d'água se tornar irrecuperável após a remoção de k arestas. 82

SUMÁRIO

1 INTRODUÇÃO 11
1.1 CONCEITOS BÁSICOS 13
1.2 TRABALHOS RELACIONADOS 16
1.3 CONTRIBUIÇÕES 18
1.4 ORGANIZAÇÃO DO TRABALHO 19

2 MARCAS D'ÁGUA DE SOFTWARE 21
2.1 MARCA D'ÁGUA ESTÁTICA × MARCA D'ÁGUA DINÂMICA . . . 24
2.2 MARCA D'ÁGUA × IMPRESSÃO DIGITAL 24
2.3 ATAQUES 25
2.4 MARCAS D'ÁGUA DE SOFTWARE BASEADAS EM GRAFOS . . . 26

3 MARCA D'ÁGUA DE CHRONI E NIKOLOPOULOS 28
3.1 A MARCA D'ÁGUA DE CHRONI E NIKOLOPOULOS 28
3.1.1 Propriedades estruturais 31
3.2 GRAFOS DE PERMUTAÇÃO REDUTÍVEIS CANÔNICOS 36
3.3 NOVO ALGORITMO DE DECODIFICAÇÃO 47
3.4 DECODIFICAÇÃO LINEAR ($K \leq 2$ ARESTAS REMOVIDAS) 52
3.4.1 Recuperação do caminho hamiltoniano 52
3.4.2 Determinando o elemento fixo 66
3.4.3 Determinação dos filhos da raiz 70
3.4.4 Recuperando arestas removidas 73
3.5 DECODIFICAÇÃO EM TEMPO POLINOMIAL (K ARESTAS REMOVIDAS) 74
3.6 RESULTADOS COMPUTACIONAIS 77
3.6.1 Comparativo dos algoritmos de decodificação 78
3.6.2 A resiliência da marca d'água de Chroni e Nikolopoulos . . . 79
3.7 CONCLUSÃO 82

4 MARCA D'ÁGUA RANDOMIZADA 84
4.1 A MARCA D'ÁGUA RANDOMIZADA 85
4.2 IMPLEMETAÇÃO DA CODIFICAÇÃO EM TEMPO LINEAR 89
4.3 RESILIÊNCIA A ATAQUES DE DISTORÇÃO 93
4.4 PROPRIEDADES DA MARCA D'ÁGUA RANDOMIZADA 97
4.4.1 Diversidade 97
4.4.2 Resiliência 98

4.4.3 Taxa de dados 98
4.4.4 Furtividade 99
4.5 CONCLUSÃO 100

5 CONCLUSÕES 101

REFERÊNCIAS 104

1 INTRODUÇÃO

A reprodução ilegal tem sido um dos maiores problemas enfrentados pela indústria de software. De acordo com o último estudo sobre reprodução ilegal de software (pirataria) realizado pela Business Software Alliance, o valor comercial de softwares não licenciados ao redor do mundo totalizou 62.7 bilhões de dólares em 2013 [1]. No Brasil, a situação é alarmante, com uma taxa de pirataria próxima a 53%, o valor comercial de softwares não licenciados é superior a 2,8 bilhões de dólares [2].

Diante deste cenário, os proprietários de software estão utilizando cada vez mais medidas legais e diversas abordagens técnicas para inibir a pirataria de software. No que tange as medidas legais, no contexto de software, existem os registros, as leis de direitos autorais, as licenças e as patentes. Apesar de se destacarem como boas alternativas, cada medida citada apresenta alguma limitação [3]. Desta forma, os desenvolvedores ainda buscam medidas técnicas baseadas em hardware ou em software para proteger programas de sua propriedade contra pirataria.

Um método baseado em hardware tipicamente utilizado consiste de um dispositivo externo conhecido como *dongle*. Com o *dongle*, o software é executado somente se o dispositivo estiver conectado ao computador, impedindo o uso de cópias não autorizadas do software protegido. Normalmente é utilizado em aplicativos de alto custo e destinados a mercados com alto grau de especificidade ou em servidores, seja garantindo o contrato temporário de utilização do software, ou restringindo o uso de um sistema a um número limitado de usuários. Embora seja uma técnica eficiente de restrição para usuários leigos, usuários experientes conseguem perturbar o comportamento do dispositivo [4].

Métodos baseados em software incluem autenticação de código, execução de software como serviço e marca d'água. Na autenticação de código, os usuários possuem o código completo e os dados de autenticação são enviados pela rede. Já na execução de software como serviço, o usuário não possui o código final, a execução de todo ou parte do software é realizada em um servidor remoto. A técnica de marca d'água insere no programa uma informação secreta, chamada de marca d'água de software, como prova de sua autoria ou propriedade.

A técnica de marca d'água difere de outras técnicas antipirataria de maneira significativa. Em primeiro lugar, ao contrário do *dongle* e técnicas relacionadas, as marcas d'água não dependem da existência de um hardware especializado e seguro. E diferente da autenticação de código e da execução de software como serviço, o uso de marca d'água não requer o tráfego de informações adicionais pela rede para que o programa protegido seja executado, já que a informação de identificação de autoria e/ou propriedade está inserida no próprio código do programa. Além disso, a técnica de marca d'água não impede que um programa seja distribuído ilegalmente, mas fornece meios para que seja possível identificar o responsável pela distribuição do programa, tornando-se aplicável em diversos cenários [5].

Neste contexto, o presente trabalho tem por objetivo realizar um estudo detalhado de técnicas de marca d'água existentes na literatura e apresentar uma forma eficiente e eficaz de codificar uma dada informação de identificação em uma marca d'água de software que será inserida no programa a ser protegido, fornecendo proteção antipirataria.

Em nossos estudos, consideramos um esquema de marca d'água baseado em grafos, onde um identificador — que por simplicidade será definido como um inteiro — é codificado na forma de um grafo especial, que poderá ser representado estaticamente em uma linguagem de programação qualquer ou dinamicamente por meio

da construção do grafo marca d'água em um determinado instante da execução do programa.

1.1 Conceitos Básicos

Um grafo direcionado (chamado apenas de grafo) G consiste de um conjunto finito não vazio $V(G)$ de elementos chamados vértices e um conjunto finito $E(G)$ de pares ordenados de vértices chamados arestas. Para cada aresta (u, v) dizemos que u é o vértice cabeça e v é a cauda. Também dizemos que a aresta (u, v) deixa u e chega em v, e que a aresta (u, v) é incidente aos vértices u e v, e que, portanto, u e v são adjacentes.

Os conjuntos $N_G^+(v)$, $N_G^-(v)$ e $N_G(v) = N_G^+(v) \cup N_G^-(v)$ são chamados de vizinhança de saída, vizinhança de entrada e vizinhança de v, respectivamente. E chamamos os vértices em $N_G^+(v)$ de vizinhos de saída, em $N_G^-(v)$ de vizinhos de entrada e em $N_G(v)$ de vizinhos de v.

Seja $v \in V(G)$, o grau de saída de v (denotado por $d_G^+(v)$) é o número de arestas em G que têm cabeça em v. De maneira análoga, o grau de entrada de v (denotado por $d_G^-(v)$) é o número de arestas em G que têm cauda em v.

Seja H um grafo. Dizemos que H é um subgrafo do grafo G se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Se H é um grafo não direcionado, dizemos que H é conexo se existe uma aresta (u, v) ou (v, u) entre qualquer par de vértices u e v. H é uma floresta se não possui ciclos. Uma floresta conexa é uma árvore.

Um passeio em G é uma sequência alternante $W = x_1a_1x_2a_2...x_{k-1}a_{k-1}x_k$ de vértices x_i e arestas a_j de G tal que a cabeça de a_i é x_i e a cauda de a_i é x_{i+1}, para

todo $i = 1, 2..., k-1$, quando os vértices e as arestas de W são distintos dizemos que W é um caminho. Se os vértices $x_1, x_2, ..., x_{k-1}$ são distintos com $x_1 = x_k$, dizemos que W é um ciclo. Um caminho ou ciclo é hamiltoniano se $V(W) = V(G)$.

Um grafo de fluxo (G, s) é um grafo G com um vértice diferenciado s, tal que todo vértice de G pode ser alcançado por s. Dizemos que um vértice u domina um vértice v se todo caminho a partir de s até v passa por u, ou seja, para visitar o vértice u é necessário primeiro visitar v.

Um grafo de fluxo de controle (CFG[1]) é uma representação gráfica das sequências de execuções possíveis de uma função [6]. O grafo é composto por blocos básicos (vértices) e arestas direcionadas. Cada bloco básico é uma sequência de instruções em que o fluxo de controle inicia no começo do bloco e só termina na instrução que encerra o bloco. Dentro de um bloco básico a transferência de controle só pode ocorrer na última instrução do bloco. Assim, todas as instruções de desvio, bem como instruções explícitas ou implícitas de salto[2], só podem aparecer como sendo a última instrução de um bloco básico. O conjunto de blocos básicos de uma sequência de instruções é construído usando o seguinte algoritmo:

1. Determine a primera instrução de cada bloco básico, chamada de líder

    - A primeira instrução de uma sequência é uma líder.
    - Qualquer instrução que seja alvo de um desvio condicional ou não condicional é uma líder.
    - Qualquer instrução seguinte a um desvio condicional ou não condicional é uma líder.

[1] Sigla de Control Flow Graph.
[2] As instruções de salto transferem o controle do programa de um lugar para outro.

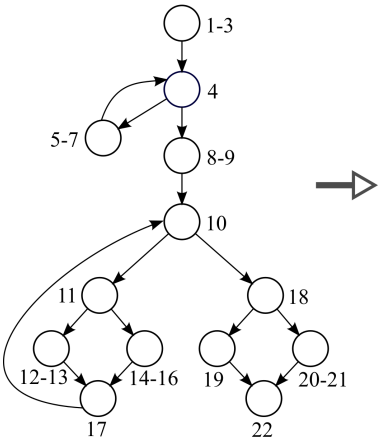
2. Para cada líder construa um bloco básico contendo todas as instruções até o próximo líder ou o fim da sequência.

O CFG é construído adicionando dois blocos especiais representando o ponto onde começa e onde termina a execução, chamados inicial e final, respectivamente, e arestas direcionadas entre os blocos básicos. Uma aresta é adicionada entre dois blocos básicos B_1 e B_2 se:

i. Há uma instrução de desvio condicional ou não condicional da última instrução em B_1 para a primeira instrução em B_2.

ii. B_2 segue imediatamente B_1 na sequência de instruções e B_1 não termina em uma instrução de desvio não condicional.

iii. A última instrução em B_1 é um salto para a primeira instrução em B_2.




```
   void main()
   {
1     int c, first, last, middle, n, search, array[100];
2     scanf("%d",&n); //number of elements
3     c = 0;
4     while (c < n){
5        scanf("%d",&array[c]);
6        c++;
7     }
8     scanf("%d",&search); //value to find
9     first = 0; last = n - 1; middle = (first+last)/2;
10    while( array[middle] != search && first <= last ){
11       if ( array[middle] < search ){
12          first = middle + 1;
13          middle = (first + last)/2;}
14       else{
15          last = middle - 1;
16          middle = (first + last)/2;}
17    }
18    if (item == a[mid]) {
19       printf("Binary search successfull!!\n")}
20    else {
21       printf("Search failed! \n")}
22 }
```

Figura 1.1: Construção do CFG

Podemos observar que o CFG apresentado à esquerda da Figura 1.1, correspondente ao código apresentado à direita, foi construído de acordo com o algoritmo apresentado acima.

Definimos como análise estática o método de depuração de programas feito através da análise do código sem realizar a execução do programa, podendo utilizar ferramentas, como o grafo de fluxo do programa, por exemplo. Já a análise dinâmica, envolve a execução do programa e a observação de seu comportamento.

1.2 Trabalhos Relacionados

O uso de marcas d'água em papel para evitar falsificação data do século $XIII$. Em meados de 1990, Davidson e Myrhvold [7] utilizaram este antigo conceito no contexto de proteção de software, propondo a primeira marca d'água de software cujo intuito era impedir – ou pelo menos desencorajar – a pirataria de software. A ideia era embarcar a marca d'água através da reorganização dos blocos básicos do grafo de fluxo de controle. Este esquema de marca d'água pode ser facilmente atacado, sendo necessário apenas uma reordenação aleatória dos blocos básicos do programa.

Qu e Potkonjak [8] propuseram embarcar a marca d'água na alocação de registradores do programa. Como toda marca d'água baseada na mudança da estrutura do programa, esse esquema é muito frágil, pois assim como o proprietário do programa pode mudar a estrutura do programa para embarcar a marca d'água, o atacante também pode modificá-la para remover a marca dágua. Myles e Collberg [9] implementaram o algoritmo de Qu e Potkonjak, apontaram que não é possível obter confiança no reconhecimento da marca d'água e propuseram uma melhoria no algoritmo de modo a permitir uma extração robusta da marca d'água descon-

siderando cenários de ataque. Zhu e Thomborson [10] discutiram alguns aspectos relacionados aos algoritmos de Qu e Potkonjak e de Myles e Collberg. Outros trabalhos que discutem esquemas de marca d'água baseados em alocação de registradores incluem [11, 12].

Stern *et al.* [13] apresentaram uma técnica de espalhamento espectral[3] para embarcar a marca d'água. O esquema embarca a marca d'água substituindo certas sequências de instruções por outras sequências diferentes, mas semanticamente equivalentes. Collberg e Sahoo [14] implementaram o esquema de Stern *et al.* e analisaram o comportamento da marca d'água diante de vários tipos de ataque. Este esquema de marca d'água inspirou outros trabalhos, como de Zhang *et al.* [15] e Feng *et al.* [16].

Monden *et al.* [17] propuseram inserir a marca d'água em instruções que nunca são executadas e guardar essas instruções em predicados opacos [4]. Fukushima e Sakurai [18] e Arboit *et al.* [19] apresentaram melhorias para o esquema de marca d'água proposto por Monden *et al.*. Collberg e Myles [20] implementaram estes algoritmos na ferramenta Sandmark.

Cousot e Cousot [21] apresentaram um algoritmo de interpretação abstrata[5] para embarcar a marca d'água em valores atribuídos a variáveis inteiras locais indicadas durante a execução do programa. Estes valores podem ser determinados por meio de uma análise do programa sob o ponto de vista de interpretação abstrata, permitindo detectar a marca d'água mesmo que possua apenas parte do programa

[3]No contexto de marca d'água, as técnicas de espalhamento espectral representam os dados de um documento como um vetor e modifica cada componente do vetor com um valor aleatório pequeno, que corresponde a marca d'água.

[4]Predicado opaco é uma expressão que avalia "true" ou "false" com seu resultado conhecido *a priori* pelo sistema ou programador que a introduziu.

[5]Neste contexto, interpretação abstrata é um técnica que tenta aproximar uma análise de software básica (no máximo, executando o programa) de uma análise refinada, sacrificando precisão em prol do tempo de execução.

com a marca d'água.

O primeiro esquema de marca d'água concentrado em explorar os conceitos de Teoria de Grafos foi formulado por Venkatesan, Vazirani e Sinha em 2001 [22]. Esta técnica, que foi mais tarde patenteada [23], toma um inteiro e o codifica na forma de um grafo direcionado especial correspondente ao grafo marca d'água. A marca d'água é embarcada no programa com o uso de predicados opacos, que são responsáveis pela adição de arestas "falsas" no CFG, e cada bloco básico pertencente a marca d'água é marcado para possibilitar sua extração.

Collberg *et al.* apresentaram diversos algoritmos para codificar marcas d'água baseadas em grafos e embarcá-las no programa a ser protegido [24, 25, 26, 20, 5, 27, 28], além de alguns modelos de ataques destinados a danificar ou desativar a marca d'água. Além disso, os autores descreveram algumas classes de grafos que poderiam ser utilizadas para codificar marcas d'água [20], dentre as quais destaca-se a classe dos grafos de permutação redutíveis (RPG) que é uma subclasse dos grafos de fluxo redutíveis (subclasse dos CFGs [6]).

Posteriormente, Chroni e Nikolopoulos, em uma série de trabalhos [29, 30, 31, 32, 33, 34, 35], apresentaram um algoritmo linear de codificação de marca d'água que gera grafos pertencentes a classe RPG.

1.3 Contribuições

Nesta tese, fornecemos uma caracterização formal para a classe dos grafos gerados pelo codec de Chroni e Nikolopoulos proposto no trabalho "An efficient graph codec system for software watermarking" [36, 37, 38], um algoritmo de decodificação dos grafos obtidos por este codec mais simples que o algoritmo apresentado por estes

autores [36, 37] e um algoritmo polinomial para recuperar a marca d'água, sempre que possível, mesmo com um número k constante de arestas removidas [39, 37]. Provamos que é possível detectar $k \leq 5$ modificações de arestas (remoções e adições) em tempo polinomial e que sempre é possível recuperar uma marca d'água que sofreu a remoção de $k \leq 2$ arestas. E apresentamos um algoritmo linear que sempre recupera o grafo original após a remoção de $k \leq 2$ arestas [36, 40].

Além disso, estudamos a resiliência[6] do esquema considerado diante de ataques de distorção[7] e apresentamos resultados computacionais que evidenciam que a probabilidade de uma marca d'água se tornar irrecuperável pela remoção de um número k fixo de arestas tende a zero a medida em que o tamanho da marca d'água aumenta [39].

Considerando algumas característica desejáveis discutidas no Capítulo 2, propomos um esquema de marca d'água de software baseado em grafos com implementação em tempo linear, que consegue, através de randomização, codificar uma dada informação de identificação em vários grafos marca d'água distintos, tornando mais difícil sua localização e remoção por parte do atacante [41]. Além disso, utilizamos técnicas de detecção/correção de erros para prover resiliência à marca d'água, de modo que nosso codec apresenta resiliência parametrizável.

1.4 Organização do Trabalho

O trabalho está organizado da seguinte forma:

[6]Esta propriedade será discutida no Capítulo 2, refere-se a capacidade da marca d'água ser recuperada após sofrer modificações.

[7]Ataque no qual o atacante altera sintaticamente o programa por meio de transformações preservadoras de semântica, incapacitando a recuperação da marca d'água pela função extratora.

Capítulo 2: Apresentamos alguns conceitos básicos relacionados à marca d'água de software, em especial marcas d'água de software baseadas em grafos.

Capítulo 3: Recordamos o esquema de marca d'água baseada em grafos proposto por Chroni e Nikolopoulos [33] listando algumas propriedades estruturais. Definimos e caracterizamos a classe dos grafos de permutação redutíveis canônicos, que corresponde às marcas d'água produzidas por Chroni e Nikolopoulos. Fornecemos um algoritmo que sempre recupera o identificador mesmo após a realização de um ataque à marca d'água que tenha causado a remoção de até duas arestas e algoritmos que recuperam até k arestas removidas, sempre que possível. Além disso, apresentamos alguns resultados computacionais que comprovam a eficiência do algoritmo de decodificação proposto, e ajuda a compreender a resiliência do esquema de Chroni e Nikolopoulos.

Capítulo 4: Enfatizamos algumas características desejáveis aos esquemas de marca d'água de software e que, de alguma forma, não foram atendidas pelo codec de Chroni e Nikolopoulos, e propomos um novo esquema de marca d'água que utiliza randomização para gerar grafos distintos para uma mesma chave e códigos de detecção e correção de erros para recuperação de ataques.

Capítulo 5: Apresentamos algumas conclusões e considerações finais.

2 MARCAS D'ÁGUA DE SOFTWARE

Marca d'água de software é um método de proteção contra pirataria, no qual uma informação secreta, também chamada de identificador ou chave, é inserida no programa para uma posterior comprovação de sua autoria ou propriedade. O detentor dos direitos autorias pode estabelecer a propriedade do software extraindo esta informação secreta de uma cópia não autorizada.

Um esquema de marca d'água pode ser formalmente descrito pelas funções a seguir [27]:

$$embarcador(P, \omega, k) \rightarrow P_\omega$$

$$extrator(P_\omega, k) \rightarrow \omega$$

onde o *embarcador* transforma um programa P em um programa P_ω que possui a marca d'água ω embarcada usando o parâmetro secreto k, e o *extrator* extrai ω de P_ω. O código do programa alterado P_ω deve satisfazer as seguintes propriedades:

- P_ω possui as mesmas características funcionais de P;
- a partir de quase todo código P'_ω "suficientemente próximo" de P_ω, é possível recuperar a informação ω, uma vez que se conheça k.

As definições apresentadas acima podem apresentar algumas variações, por exemplo, o *extrator* pode retornar uma lista de marcas d'água ω e um nível de confiança para cada marca d'água retornada. A função *extrator* pode ainda classificar um esquema de marca d'água em cego ou informado [5]. Em um esquema de marca

d'água cedo, a função *extrator* recebe como entrada apenas o programa marcado e o parâmetro secreto k. Em um esquema informado a função *extrator* também tem acesso ao programa original (sem a marca d'água) e/ou a própria marca d'água.

Uma vez que uma marca d'água de software é embarcada em um software que será distribuído aos usuários, tal software pode ser analisado e atacado por um adversário. Assim, é essencial modelar corretamente as metas e capacidades do atacante. Em particular, um atacante poderá impedir que os objetivos do defensor sejam atingidos por meio das seguintes funções:

$$detecta(P_\omega) \rightarrow [0.0, 1.0]$$

$$ataque(P_\omega) \rightarrow P'_\omega.$$

A função *detecta* pode ser vista como uma forma de modelar confidencialidade, pela qual sabemos se um atacante é capaz de detectar ou não a marca d'água, ou seja, quanto mais próximo de 1 for o valor de *detecta* maior é a probabilidade do atacante localizar a marca d'água. Enquanto a função *ataque* pode ser vista como uma forma de modelar a disponibilidade da marca d'água, por meio da qual sabemos se um atacante é capaz de atacar um objeto que suspeita ser a marca d'água a ponto de torná-la irreconhecível e não ser possível extraí-la (desconsiderando mudanças severas que podem tornar o programa inutilizável).

Definimos quatro características desejáveis a um esquema de marca d'água, a saber: furtividade, resiliência, alta taxa de dados e diversidade. Um esquema de marca d'água furtivo é aquele que resiste as tentativas do atacante de descobrir a posição em que a marca d'água foi inserida no programa, isto é, resiste as tentativas do atacante de descobrir uma função *detecta*. Considerando essa característica, quando não possuímos um modelo preciso (e tratável) do atacante, é difícil definir

uma métrica exata para avaliá-la. Em vez disso, vamos avaliá-la somente em termos qualitativos.

Um esquema de marca d'água resiliente é aquele que resiste as tentativas do atacante de descobrir uma função de *ataque*. De um modo geral, vamos considerar que uma marca d'água é resiliente quando um atacante consegue descobrir uma função de *ataque* e, mesmo assim, é possível recuperar o identificador a partir da marca d'água danificada.

Outra característica associada a marca d'água é chamada de taxa de dados, que expressa a eficiência de espaço do esquema de marca d'água. Há duas variantes: a taxa de dados dinâmica e taxa de dados estática. Em ambas as variantes, o numerador é o número de bits na marca d'água, e o denominador mede a sobrecarga (em bytes) adicionada pela marca d'água. Na medida estática, calculamos o aumento do tamanho do programa compilado quando acrescentamos a marca d'água. Na medida dinâmica, calcula-se o aumento no consumo de espaço de armazenamento, tal como uma função do número de bits da marca d'água quando o programa é executado. Em outras palavras, desejamos codificar o maior número de informação possível em uma marca d'água utilizando poucos bits (a marca d'água deve possuir um tamanho pequeno).

E, por fim, chamamos a última característica desejável de diversidade, que é a capacidade de codificar um mesmo identificador em marcas d'água distintas, possibilitando ao desenvolvedor gerar versões distintas de um mesmo programa marcado, aumentando a dificuldade de localização da marca d'água por parte do atacante.

Um esquema de marca d'água ideal teria furtividade, resiliência, diversidade e alta taxa de dados. Na prática, não é possível maximizar qualquer uma destas características sem ter algum efeito colateral sobre uma ou mais das outras ca-

racterísticas desejáveis. Por exemplo, aumentar o tamanho da marca d'água para melhorar a resiliência pode comprometer a furtividade, porque a adição de mais "código marca d'água" pode fornecer mais pistas da localização da marca d'água ao atacante.

2.1 Marca d'água estática × Marca d'água dinâmica

Os esquemas de marca d'água podem ser classificados de acordo com a técnica utilizada na extração, podendo ser estáticos ou dinâmicos [5]. Uma marca d'água de software estática pode ser inserida na área de dados do programa (chamada de marca d'água de dados) ou no texto do código do pragrama (chamada de marca d'água de código). A extração de uma marca d'água estática é realizada sem a necessidade de executar o programa [27].

Uma marca d'água dinâmica é inserida no estado de execução do programa, isto é, em um esquema de marca d'água dinâmico, a informação embarcada não corresponde a própria marca d'água, e sim a trechos de código que geram a marca d'água desejada quando o programa é executado [28].

2.2 Marca d'água × Impressão digital

Num cenário de marca d'água, todas as cópias distribuídas de um determinado programa possuem apenas uma informação de identificação, correspondente ao identificador do proprietário do programa. Enquanto num cenário de impressão digital (*fingerprint*), em cada cópia do programa é embarcado um identificador diferente, correspondente ao usuário que adquiriu tal cópia. Em outras palavras, cada usuário recebe uma versão única do programa, e como resultado, o proprietário pode

rastrear o usuário responsável por uma possível distribuição ilegal de seu programa [5].

2.3 Ataques

Como em todo cenário de segurança, devemos considerar as situações em que a marca d'água pode ser modificada, removida ou não identificada, impossibilitando a recuperação do identificador. Como descrito na literatura, uma marca d'água pode sofrer diversos tipos de ataque [5], dentre os quais podemos listar os seguintes:

Ataques de adição: o atacante insere uma nova marca d'água no programa marcado, para que o proprietário original do programa não possa provar a propriedade com sua marca d'água original;

Ataques de subtração: o atacante consegue localizar a marca d'água e a remove completamente sem afetar a funcionalidade do programa;

Ataques de distorção: são aplicadas transformações que preservam a semântica do programa (otimização de código, ofuscação[1], entre outros) a fim de danificar a marca d'água e impossibilitar a recuperação do identificador;

Ataques de reconhecimento: o atacante modifica ou desativa o algoritmo detector da marca d'água (correspondente a função *extrator*), ou suas entradas, de modo que o resultado obtido seja falso. Por exemplo, em uma disputa judicial, o atacante pode afirmar que seu algoritmo detector é o que deve ser utilizado para comprovar a propriedade do programa.

[1] A ofuscação de código consiste na realização de alterações no código de um programa para torná-lo mais difícil de ser compreendido.

2.4 Marcas d'água de software baseadas em grafos

Num esquema de marca d'água de software baseado em grafos, a marca d'água é codificada na forma de um grafo G que pertence a uma classe especial de grafos. Há basicamente dois tipos de marca d'água de software baseada em grafos: as baseadas no algoritmo de Venkatesan, Vazirani e Sinha [22], que podem ser definidas como estáticas; e as baseadas no algoritmo de Collberg e Thomborson [28], que podem ser definidas como dinâmicas.

Nos esquemas de marca d'água de software baseados em grafos ditos estáticos, o grafo marca d'água é embarcado no grafo de fluxo de controle do programa que pretende-se proteger. A inclusão pode ser realizada por meio da adição de arestas no grafo de fluxo original [22], ou por meio da inserção de código *dummy* que corresponde ao grafo marca d'água [32, 33, 42]. Há usualmente quatro algoritmos nestes esquemas:

- *codificador:* converte o identificador id em um grafo ω — a marca d'água;
- *decodificador:* extrai id a partir da marca d'água ω;
- *embarcador:* recebe como parâmetro o programa P (podendo ser o código binário ou o código-fonte em alguma linguagem de programação), a marca d'água ω e uma possível chave secreta k, e retorna como saída um programa marcado P_ω. O embarcador preserva a semântica do programa, isto é, P e P_ω tem o mesmo comportamento de entrada e saída; e
- *extrator:* retorna ω a partir de P_ω.

Em particular, chamamos o par *(codificador, decodificador)* simplesmente de

codec. Do ponto de vista de Teoria de Grafos, estamos procurando uma classe de grafos $\mathcal{G}$ e um codec correspondente com as seguintes propriedades:

Grafos apropriados: Os grafos em $\mathcal{G}$ devem ser direcionados com uma ordenação em suas arestas, isto é, qualquer vértice de $\mathcal{G}$ é alcançado por um vértice diferenciado, chamado raiz. Além disso, os grafos em $\mathcal{G}$ devem possuir grau de saída máximo baixo, para que possam coincidir com grafos de fluxo gerados a partir de programas reais;

Alta resiliência: O algoritmo *decodificador* deve ser capaz de recuperar o identificador a partir do grafo marca d'água G, mesmo após pequenas perturbações de G (que podem ser causadas pela ação de um atacante), tais como inclusões e remoções de um número constante de vértices e/ou arestas;

Diversidade: No contexto de marca d'água, a diversidade refere-se ao fato de ser possível codificar um identificador em vários grafos marca d'água;

Tamanho pequeno: $|P_\omega| - |P|$ deve ser pequeno;

Eficiência: O codec deve ser implementado de modo eficiente, preferencialmente em tempo linear.

Nos algoritmos dinâmicos, a marca d'água é gerada durante a execução do programa, por meio de três passos básicos. Primeiro é escolhido um grafo G apropriado para representar a marca d'água, e G é particionado em alguns subgrafos. Depois contrói-se um conjunto de códigos para geração de cada subgrafo e insere-se estes códigos ao longo de um caminho de execução especial, que é executado quando uma chave secreta é fornecida como entrada para o programa [28].

3 MARCA D'ÁGUA DE CHRONI E NIKOLOPOULOS

Neste capítulo, descreveremos o algoritmo de codificação de marca d'água proposto por Chroni e Nikolopoulos [30, 32, 33, 29] utilizado como referência na primeira parte de nossos estudos. Além disso, apresentaremos os resultados inicialmente obtidos, que são: prova de algumas propriedades estruturais observadas na construção da marca d'água, caracterização da classe dos grafos de permutação redutíveis canônicos (grafos gerados pelo algoritmo de codificação de Chroni e Nikolopoulos), algoritmo linear para reconhecimento dos grafos desta classe, algoritmo linear para decodificação da marca d'água, algoritmo polinomial para recuperação da marca d'água após ataques de distorção com remoção de até duas arestas, algoritmo polinomial que recupera, sempre que possível, uma marca d'água após a remoção de um número k (constante) de arestas e alguns resultados computacionais.

3.1 A Marca d'água de Chroni e Nikolopoulos

Revisitamos agora a codificação proposta por Chroni e Nikolopoulos [32] para marcas d'água baseadas em grafos. Para compor o grafo marca d'água, a técnica utiliza algumas sequências numéricas especiais que serão definidas mais adiante, e, sem perda de generalidade, vamos considerar que o índice do primeiro elemento em toda sequência considerada é 1.

Sejam ω um inteiro positivo, e n o tamanho da representação binária B de ω. Sejam n_0 o número de 0's e n_1 o número de 1's em B e seja f_0 o índice do 0 mais à esquerda em B. O binário estendido B^* é obtido pela concatenação de n dígitos

1, seguido do complemento de um de B (isto é, B com seus dígitos invertidos), e um dígito 0 adicional à direita. Denotamos por $n^* = 2n + 1$ o tamanho de B^*, e definimos $Z_0 = (z_i^0), i = 1, ..., n_1 + 1$, como a sequência ascendente de índices de 0's em B^*, e $Z_1 = (z_i^1), i = 1, ..., n + n_0$, como a sequência ascendente de índices de 1's em B^*.

Considere como exemplo a chave $\omega = 37$, da qual temos $B = 100101$, $n = 6$, $n_0 = 3$, $n_1 = 3$, $B^* = 1111110110100$, $n^* = 13$, $Z_0 = (7, 10, 12, 13)$ e $Z_1 = (1, 2, 3, 4, 5, 6, 8, 9, 11)$.

Seja S uma sequência de inteiros. Denotamos por S^R a sequência formada pelos elementos de S em ordem inversa. Se $S = (s_i)$, para $i = 1, ..., t$ e há um inteiro $k \leq t$, tal que, a subsequência composta pelos elementos de S com índice menor ou igual a k é ascendente, e a subsequência de elementos de S com índices maiores ou iguais a k é descendente, dizemos que S é *bitônica*. Se todo elemento t de uma sequência S são distintos e pertencem a $\{1, ..., t\}$, então S é uma *permutação*. Se S é uma permutação de tamanho t, e para todo $1 \leq i \leq t$ a igualdade $i = s_{s_i}$ é satisfeita, dizemos que S é *auto-inversível*. Neste caso, os pares ordenados (i, s_i) são chamados *2-ciclo* de S, se $i \neq s_i$, e *1-ciclo* de S, se $i = s_i$. Se S_1, S_2 são sequências (respectivamente, caminhos em um grafo), denotamos por $S_1||S_2$ a sequência (respectivamente, caminho) formado pelos elementos de S_1 seguidos dos elementos de S_2.

Retornando ao algoritmo de Chroni e Nikolopoulos, definimos $P_b = (b_i)$, com $i = 1, ..., n^*$, como a permutação bitônica $Z_0||Z_1^R$. Finalmente, a *permutação auto-inversível* $P_s = (s_i)$ é obtida a partir de P_b da seguinte forma: para $i = 1, \ldots, n^*$, o elemento s_{b_i} assume o valor b_{n^*-i+1}, e o elemento $s_{b_{n^*-i+1}}$ assume o valor b_i. Em outras palavras, os n pares não-ordenados de elementos distintos $(p, q) = (b_i, b_{n^*-i+1})$ de P_b, para $i = 1, \ldots, n$, são os chamados 2-*ciclos* de P_s, com p ocupando a posição

q em P_s, e vice-versa. Como o índice $i = n+1$ é a solução da equação $n^* - i + 1 = i$, o elemento central b_{n+1} de P_b aparecerá em P_s com índice igual ao próprio b_{n+1}, constituindo assim o único 1-*ciclo* de P_s, que é também chamado de *elemento fixo* de P_s, denotado por f. Assim, voltando ao exemplo com $\omega = 37$, $Z_0 = (7, 10, 12, 13)$ e $Z_1 = (1, 2, 3, 4, 5, 6, 8, 9, 11)$, temos $P_b = (7, 10, 12, 13, 11, 9, 8, 6, 5, 4, 3, 2, 1)$ e $P_s = (7, 10, 12, 13, 11, 9, 1, 8, 6, 2, 5, 3, 4)$.

A marca d'água gerada pelo algoritmo de codificação de Chroni e Nikolopoulos [32] é um grafo direcionado G, tal que, $V(G) = \{0, 1, ..., 2n + 2\}$ e $E(G)$ possui $4n + 3$ arestas, a saber: *arestas de caminho* $(u, u - 1)$, para todo $u = 1, \ldots, 2n + 2$, produzindo um caminho hamiltoniano que é único em G; e *arestas de árvore* de u para $q(u)$, $u = 1, \ldots, n^*$, onde $q(u)$ é definido como o vértice $v > u$ à esquerda de u e o mais próximo possível de u em P_s, se tal v existir, ou $2n + 2$, caso contrário. Um grafo assim obtido é um tipo especial de grafo de permutação redutível chamado *grafo de permutação redutível canônico*, cuja caracterização formal será apresentada na Seção 3.2. O vértice 0 é o único sorvedouro (vértice com grau de saída igual a zero) de G.

Retomando o exemplo, a marca d'água associada a $\omega = 37$ terá, ao longo de seu caminho hamiltoniano, as arestas $14 \to 13 \to \ldots \to 0$, e apresentará também as arestas de árvore $(1, 9), (2, 6), (3, 5), (4, 5), (5, 6), (6, 8), (7, 14), (8, 9), (9, 11), (10, 14)$, $(11, 13), (12, 14)$ e $(13, 14)$, como ilustrado pela Figura 3.1.

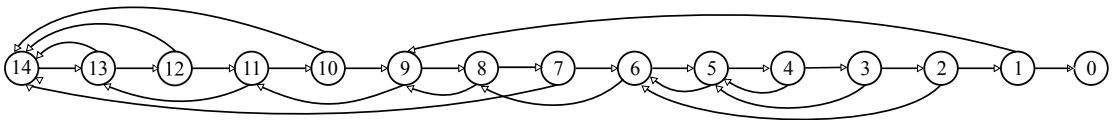


Figura 3.1: Marca d'água para a chave $\omega = 37$.

3.1.1 Propriedades estruturais

A seguir listamos algumas propriedades que identificamos ao analisar o esquema de marca d'água proposto por Chroni e Nikolopoulos e as permutações especiais associadas. Este conjunto de propriedades básicas serve de suporte para a caracterização da classe dos grafos de permutação redutíveis canônicos, que é dada na Seção 3.2, e os algoritmos descritos na Seção 3.4.

Para todas as propriedades listadas abaixo, considere um grafo marca d'água G associado a um identificador ω de tamanho n, e seja P_b e P_s, respectivamente, as permutações bitônica e auto-inversível geradas durante a construção de G.

Propriedade 1 *Para $1 \leq i \leq n$, o elemento b_{n+i+1} em P_b é igual a $n - i + 1$, isto é, os n elementos mais à direita em P_b, da direita para a esquerda, são $1, ..., n$.*

Prova: Quando percorremos os elementos de P_b da direita para esquerda, os n elementos mais à direita correspondem aos n primeiros elementos em Z_1, isto é, os índices de B^* onde um dígito 1 está localizado. Como B^* começa com uma sequência de n 1's contíguos, a propriedade segue. □

Propriedade 2 *Os elementos de índices $1, ..., n$ em P_s são todos maiores que n.*

Prova: Por construção, os dígitos de índice $1, ..., n$ em B^* são 1. Os n elementos mais à direita em P_b (isto é, elementos de índice $n+2 \leq i \leq n^*$ em P_b) correspondem aos n primeiros elementos em Z_1, ou seja, os n primeiros índices de 1's em B^*, que serão os elementos do conjunto $S = \{1, ..., n\}$. Em outras palavras, se $s \in S$, então s terá índice $n^* - s + 1 > n + 1$ em P_b. Como os elementos em P_b são todos definidos por pares que determinam suas posições em P_s, o elemento s será par do elemento

q, onde o índice $n^* - (n^* - s + 1) + 1 = s$. Como $s \leq n$, onde q claramente não pertence a S, então $q > n$. Agora, como s assumirá índice q em P_s, o elemento com índice s em P_s será par de $q > n$, o que conclui a prova. □

Propriedade 3 *O elemento fixo f satisfaz $f = n + f_0$, a menos que o identificador ω seja igual a $2^k - 1$ para algum inteiro k, quando $f = n^* = 2n + 1$.*

Prova: A permutação bitônica P_b é construída de maneira que seu $(n + 1)$-ésimo elemento $f = b_{n+1}$ é ou:

(i) o $(n + 1)$-ésimo elemento de Z_0, caso B^* tenha ao menos $n + 1$ dígitos 0; ou
(ii) o $(n + 1)$-ésimo elemento de Z_1, caso contrário.

Por construção, o número de 0's em B^* é uma unidade maior que o número de 1's em B. Se (i) é válido, então B corresponde a um identificador ω que é predecessor de uma potência de 2, implicando que todos os n dígitos de B são 1's. Se este é o caso, então a propriedade desejada segue imediatamente, uma vez que o $(n + 1)$-ésimo elemento de Z_0 será o índice do $(n + 1)$-ésimo — isto é, o último — dígito 0 em B^*. Tal índice é n^*, por construção.

Se (ii) é válido, então f é o índice do $(n + 1)$-ésimo dígito 1 em B^*. Por construção, os n primeiros dígitos 1 em B^* ocupam posições com índices $1, ..., n$, e o $(n + 1)$-ésimo dígito 1 em B^* corresponde ao primeiro dígito 1 no complemento de um de B. Como esse dígito tem índice f_0 no complemento de um de B, e como em B^* há exatamente n dígitos à esquerda do complemento de um de B, a propriedade segue. □

Propriedade 4 *Na permutação auto-inversível P_s, os elementos de índice $1, ..., f - n - 1$ são, respectivamente iguais, a $n + 1, n + 2, ..., f - 1$, e os elementos de índice $n + 1, n + 2, ..., f - 1$ são, respectivamente, iguais a $1, ..., f - n - 1$.*

Prova: Pela construção de P_s e pela Propriedade 1, segue que os elementos que ocupam as posições de índice $1, ..., n$ em P_s são os n primeiros elementos em P_b. Acontece que os $n_1 + 1$ primeiros elementos em P_b correspondem a Z_0, isto é, índices de 0's em B^*. Sabemos que o último dígito em B^* — o de índice n^* — é sempre 0. Além disso, temos que os outros dígitos 0 em B^* têm índice $z = n + d$, onde cada d é o índice de um dígito 1 em B (o binário original que representa o identificador ω). Enquanto o primeiro dígito em B é sempre 1, é também verdade que:

(i) os $f_0 - 1$ primeiros dígitos em B constituem uma sequência de 1's, quando há ao menos um 0 em B; ou
(ii) todos os n dígitos de B são 1's, quando ω é predecessor de uma potência de 2.

Qualquer que seja o caso, a Propriedade 3 nos permite afirmar que existe uma sequência de $f - n - 1$ dígitos 1 em B iniciando no primeiro dígito. Tal sequência vai aparecer em B^*, iniciando no índice $n + 1$, de tal forma que os $f - n - 1$ primeiros elementos de Z_0 serão $n + 1, n + 2, ..., +(f - n - 1) = f - 1$. Estes elementos, como podemos ver, serão exatamente os primeiros elementos em P_b. Como não há mais do que n elementos, eles serão pareados com os elementos $1, 2, ..., f - n - 1 \leq n$ (localizados da direita para a esquerda em P_b) a fim de determinar a sua posição em P_s, e a propriedade segue. □

Propriedade 5 *O primeiro elemento em P_s é $s_1 = n + 1$, e o elemento central em P_s é $s_{n+1} = 1$.*

Prova: Se o identificador ω não é predecessor de uma potência de 2, então sua representação binária B, cujo primeiro dígito é sempre 1, contêm algum dígito 0. Em virtude disso, a Propriedade 3 implica que $f \geq n + 2$ para todo inteiro ω, e a primeira igualdade segue da Propriedade 4. A segunda igualdade é garantida pela auto-inversibilidade de P_s, pela qual $s_j = u \Leftrightarrow s_u = j$. □

Propriedade 6 *Se $f \neq n^*$, então o elemento de índice n^* em P_s é igual $n_1 + 1$ e vice-versa. Se $f = n^*$, então o índice do elemento n^* em P_s também é n^*.*

Prova: Note que $f \neq n^*$ corresponde ao caso em que o identificador ω não é predecessor de uma potência de 2, isto é, $n_1 < n$. Como a sequência Z_0 tem exatamente $n_1 + 1$ elementos, onde n^* é o índice do dígito mais à direita em B^*, o elemento n^* sempre assumirá o índice $n_1 + 1$ em P_b. De acordo com o que vimos na prova da Propriedade 4, para $i \leq n$, o i-ésimo elemento em P_b também será o i-ésimo elemento em P_s, pois serão pareados com o elemento i, indexado $n^* - i + 1$ em P_b (devido a sequência inicial de n dígitos 1 em B^*). Dito isso, o elemento n^*, de índice $n_1 + 1 \leq n$ em P_b também terá índice $n_1 + 1$ em P_s. Se $f = n^*$, então a definição de f verifica a propriedade trivialmente. □

Propriedade 7 *A subsequência de P_s composta pelos elementos de índice $1, ..., n+1$ é bitônica.*

Prova: Na prova da Propriedade 4 observou-se que a subsequência composta pelos n primeiros elementos de P_s é a mesma que subsequência composta pelos n primeiros elementos de P_b. Como P_b é bitônica qualquer subsequência de P_b também é bitônica, particularmente aquela que contêm os seus n primeiros elementos. Pela Propriedade 5, o elemento central s_{n+1} de P_s é sempre igual a 1, em consequência disso, a propriedade bitônica da subsequência de elementos mais à esquerda de P_s permanece válida após seu comprimento ter crescido de n para $n+1$, cujo elemento $s_{n+1} = 1$. □

Propriedade 8 *Para $u \leq 2n$, $(u, 2n + 2)$ é uma aresta de árvore da marca d'água G se e somente se $u - n$ é o índice de um dígito 1 na representação binária B de um identificador ω representado por G.*

Prova: Os n_1+1 primeiros elementos da permutação bitônica P_b são os elementos de Z_0 que correspondem aos índices dos 0's no binário estendido B^*. Esses elementos constituem o prefixo ascendente $A = n + z_1, n + z_2, ..., n + z_{n_1}, 2n + 1$, para $i \in \{1, ..., n_1\}$ e z_i é o índice de um dígito 1 em B. A partir da prova de corretude da Propriedade 4, sabemos que para $i \leq n$, o i-ésimo elemento em P_b também será o i-ésimo elemento na permutação auto-inversível P_s. Uma vez que $n_1 \leq n$, tem-se que os n_1 primeiros elementos de P_s são precisamente os n_1 primeiros elementos de A, daí a aresta de árvore que é cauda em cada um desses elementos deve apontar, por construção, para $2n+2$. Resta mostrar que nenhum elemento $u \notin A \cup \{2n+1\}$ é cauda da aresta de árvore que aponta para $2n+2$. Mas isto pode ser visto facilmente pela Propriedade 6, já que o $(n_1 + 1)$-ésimo elemento em P_s é $2n + 1$. Uma vez que todos os vértices u com índices $i > n_1 + 1$ em P_s são certamente menores que $2n+1$, tais vértices só podem ser cauda de arestas de árvore que apontam para vértices $q(u) \leq 2n + 1$, e a prova está completa. □

Propriedade 9 *Se (u, k) é uma aresta de árvore da marca d'água G, com $k \neq 2n + 2$, então*

(i) o elemento k precede u em P_s; e

(ii) se v está localizado em algum lugar entre k e u em P_s, então $v < u$.

Prova: os dois itens são verificados trivialmente já que, por construção, toda aresta de árvore $(u, k) \in V(G)$ é tal que todo $k > u$ é o elemento que está mais perto e a esquerda de u em P_s, ou $k = 2n + 2$. □

3.2 Grafos de permutação redutíveis canônicos

Nesta seção, vamos apresentar uma caracterização para a classe dos grafos de permutação redutíveis canônicos [36]. Depois de descrever algumas terminologias e prover alguns resultados preliminares, mostraremos que os grafos pertencentes a esta classe correspondem exatamente aos grafos produzidos pelo algoritmo de codificação de Chroni e Nikolopoulos [32].

Um *grafo de fluxo redutível* [43, 44, 45] é um grafo direcionado G com uma fonte $s \in V(G)$ tal que, para cada ciclo C de G, todo caminho direcionado a partir de s até C, chega em C pelo mesmo vértice. Ou ainda, podemos dizer que G é redutível se e somente se seu conjunto de arestas pode ser dividido em dois conjuntos chamados de *arestas de avanço* e *arestas de retorno*, com as seguintes propriedades:

- as arestas de avanço formam um grafo acíclico (DAG) em que todo vértice pode ser alcançado a partir de um vértice inicial de G;
- as arestas de retorno são arestas em que a cabeça domina a cauda[1]. Estas arestas também podem ser chamadas de *arestas de ciclo*.

Sabe-se que se um grafo de fluxo redutível possuir um caminho hamiltoniano, ele é único [46]. A Figura 3.2 mostra um exemplo de grafo de fluxo redutível. As arestas sólidas e pontilhadas correspondem as arestas de avanço e de retorno, respectivamente.

[1] Um vértice m domina um vértice n em G se todo caminho em G a partir da raiz s até n contêm m

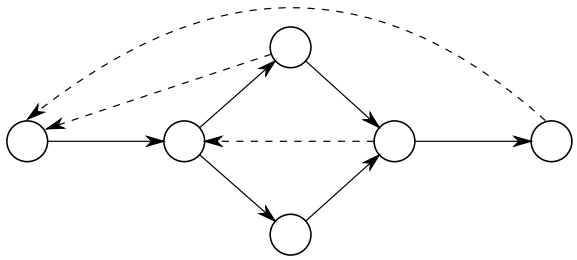


Figura 3.2: Grafo de fluxo redutível

Definição 1 *Um grafo de fluxo redutível auto-rotulável é um grafo direcionado G tal que*

(i) *G possui exatamente um caminho hamiltoniano direcionado H, onde há uma função de rotulação única $\sigma : V(G) \to \{0, 1, ..., |V(G)| - 1\}$ dos vértices de G tal que a ordem dos rótulos ao longo de H é precisamente $|V(G)|, |V(G)| - 1, ..., 0$;*

(ii) *considerando os rótulos σ como no item anterior, $N_G^+(0) = \emptyset$, $N_G^-(0) = \{1\}$, $N_G^+(|V(G)| - 1) = \{|V(G)| - 2\}, |N_G^-(|V(G)| - 1)| \geq 2$, e para todo $v \in V(G) \backslash \{0, |V(G)| - 1\}$, $N_G^+(v) = \{v - 1, w\}$, para algum $w > v$.*

A Figura 3.3 apresenta um exemplo de grafo de fluxo redutível auto-rotulável, no qual podemos observar que as propriedades *(i)* e *(ii)* da Definição 1 são satisfeitas.

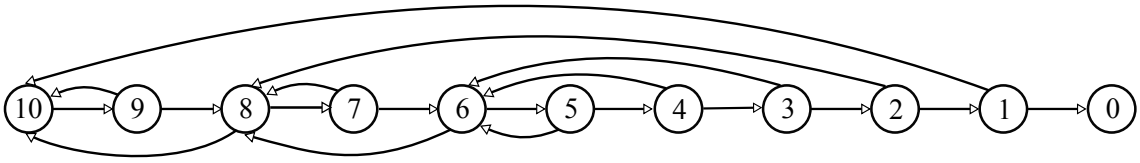


Figura 3.3: Grafo de fluxo redutível auto-rotulável

A partir de agora, sem perda de generalidade, assuma que o conjunto de vértices de qualquer grafo de fluxo redutível auto-rotulável G é dado pelo conjunto

$V(G) = \{0, 1, ..., |V(G)| - 1\}$. Ao fazer isso, podemos simplesmente comparar dois vértices, por exemplo, $v > u$ (v maior do que u), enquanto que, de outra forma, precisaríamos comparar suas imagens, por exemplo, $\sigma(v) > \sigma(u)$.

Definição 2 *Uma árvore representativa T de um grafo de fluxo redutível auto-rotulável G com caminho hamiltoniano H tem conjunto de vértices $V(T) = V(G)\backslash\{0\}$ e conjunto de arestas $E(T) = E(G)\backslash E(H)$, onde todas as arestas são não direcionadas.*

Uma árvore representativa T é uma árvore enraizada na qual a raiz é $|V(G)|-1$. Além disso, uma árvore representativa pode ser vista como uma árvore ordenada, isto é, para cada $v \in V(T)$, os filhos de v são sempre considerados de acordo com uma ordem ascendente de seus rótulos. Para $v \in T$, denotamos por $N_T^*(v)$ o conjunto de descendentes de v em T.

Observação 1 *A árvore representativa T de um grafo de fluxo redutível auto-rotulável G satisfaz a propriedade max-heap, isto é, se o vértice u é um filho do vértice v em T, então $v > u$.*

Prova: Segue do fato de T ser enraizada e pela propriedade *(ii)* da definição de grafo de fluxo redutível auto-rotulável, onde os vizinhos de entrada de w em $G\backslash E(H)$ contêm somente vértices $v < w$.□

Pelo fato de uma árvore representativa T satisfazer a propriedade max-heap, podemos dizer que T é uma árvore descendente, ordenada e enraizada. A Figura 3.4 apresenta duas árvores representativas, nas quais podemos verificar facilmente que suas arestas correspondem às arestas de retorno de grafos de fluxo redutíveis auto-rotuláveis.

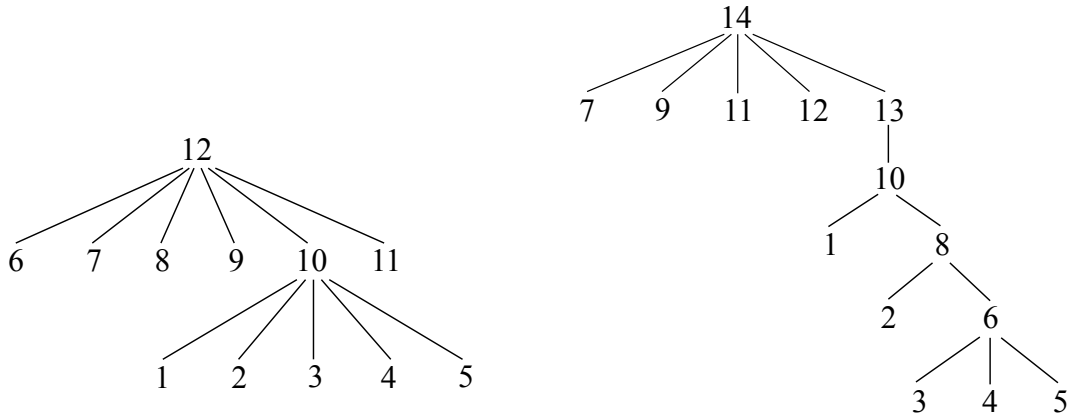


Figura 3.4: Árvores representativas

Definição 3 *Seja $S = (s_i)$, $i = 1, ..., 2n + 1$, uma permutação auto-inversível. Dizemos que S é canônica se:*

(i) há exatamente um 1-ciclo em S;
(ii) cada 2-ciclos (s_i, s_j) de S satisfaz $1 \leq i \leq n$, para $s_i > s_j$;
(iii) $s_1, ..., s_{n+1}$ é uma subsequência bitônica de S iniciando em $s_1 = n + 1$ e terminando em $s_{n+1} = 1$.

A permutação $P_s = (7, 10, 12, 13, 11, 9, 1, 8, 6, 2, 5, 3, 4)$ apresentada na Seção 3.1 é um exemplo de uma permutação auto-inversível canônica, já que satisfaz as Condições *(i), (ii), (iii)* da Definição 3.

Lema 1 *Em qualquer permutação auto-inversível canônica, o elemento fixo f satisfaz $f \in [n + 2, 2n + 1]$.*

Prova: Pela Condição *(ii)* das permutações auto-inversíveis canônicas, cada 2-ciclo de S contêm ao menos um elemento com índice i satisfazendo $1 \leq i \leq n$. Pela Condição *(i)* e dado o tamanho de S, temos que o número de 2-ciclos em S é n.

Então, pela ordem de S, cada 2-ciclo de S contem exatamente um elemento s_i com $1 \leq i \leq n$. Mas isto significa que o outro elemento em cada 2-ciclo, nomeado s_j, satisfaz $s_j \in [n+1, 2n+1]$, havendo $n+1$ valores no intervalo e somente n elementos s_j. Logo, deve haver exatamente um elemento $s_k \in [n+1, 2n+1]$ que não é parte de um 2-ciclo, e portanto, $s_k = f$. Logo, pela Condição *(iii)*, $n+1 = s_1$, por isso $f \neq n+1$, e o lema segue. □

Seja T uma árvore representativa. O percurso em pré-ordem P de T é uma sequência de seus vértices recursivamente definida como segue. Se T é vazia, P também é vazia. Caso contrário, P inicia na raiz r de T, seguida pelo percurso em pré-ordem da subárvore cuja raiz é o menor filho de r, seguido pelo percurso em pré-ordem da subárvore cuja raiz é o segundo menor filho de r, e assim por diante. O último elemento (mais à direita) de P é também referido como o elemento mais à direita de T.

Lema 2 *Um percurso em pré-ordem de uma árvore representativa T é único. Analogamente, uma árvore representativa T é unicamente definida por seu percurso em pré-ordem.*

Prova: Usaremos indução em $|V(T)|$. Se $|V(T)| \leq 1$, o lema segue trivialmente. Seja $|V(T)| > 1$, e seja v_k a folha de T unicamente definida como sendo o maior vértice entre os filhos de $v-1$, onde o caminho $v_1, ..., v_k$ da raiz v_1 de T até v_k tem os vértices v_i, para $1 < i \leq k$. Pela hipótese de indução, o percurso em pré-ordem P' de $T - \{v_k\}$ é unico. Como v_k é necessariamente o vértice mais à direita de T, o percurso em pré-ordem P de T é unicamente determinado como $P'||v_k$.

De maneira análoga, seja P um percurso em pré-ordem de alguma árvore representativa T. Se $|P| \leq 1$, não há o que provar. Por outro lado, suponha que o lema vale para um percurso em pré-ordem de tamanho $\leq k$, e considere $|P| = k$.

Seja v_k o elemento mais à direita de P. Claramente, v_k deve ser uma folha de T, e também o vértice mais à direita (isto é, maior) entre os filhos de seu pai. Agora, defina $P' = P - \{v_k\}$. Pela hipótese de indução, há uma única árvore T' cujo percurso em pré-ordem é P'. Seja v_{k-1} o elemento mais à direita de P'. Obtemos T a partir de T' fazendo v_k o filho mais à direita do menor ancestral v_j de v_{k-1} satisfazendo $v_j > v_k$, então P é claramente o percurso em pré-ordem de T. Uma vez que, não seria possível definir nenhum outro pai para v_k sem quebrar a ordem crescente de seus irmãos na árvore representativa, T é unicamente definida por P. □

O primeiro elemento do percurso em pré-ordem P de uma árvore T é sempre sua raiz. Se removemos o primeiro elemento de P, a sequência resultante é dita *percurso em pré-ordem sem raiz* de T.

Podemos agora definir a classe dos grafos de permutação redutíveis canônicos.

Definição 4 *Um grafo de permutação redutível canônico G é um grafo de fluxo redutível auto-rotulável com $2n + 3$ vértices, para algum inteiro $n \geq 1$, tal que o percurso em pré-ordem sem raiz da árvore representativa de G é uma permutação auto-inversível canônica.*

Lema 3 *Se G é uma marca d'água produzida pela codificação de Chroni e Nikolopoulos [32], então G é um grafo de permutação redutível canônico.*

Prova: Temos, pelo algoritmo de codificação de Chroni e Nikolopoulos, que a marca d'água G associada ao identificador ω cuja representação binária B tem tamanho n, é constituída de um conjunto de vértices $V(G) = \{0, ..., 2n + 2\}$ e um conjunto de arestas $E(G)$ que pode ser particionado em *arestas de caminho* e *arestas de árvore* onde todas as condições na definição de grafo redutível auto-rotulável são satisfeitas,

como pode ser facilmente verificado. Agora, pela Propriedade 9 da marca d'água de Chroni e Nikolopoulos, as arestas de árvore de G constituem uma árvore representativa T de G cujo percurso em pré-ordem sem raiz é precisamente a permutação auto-inversível P_s determinada pelo algoritmo de codificação [32] como uma função de B. Consequentemente, isto é usado para provar que P_s é canônica. A primeira condição para P_s ser canônica é apontada pela Propriedade 3 (o elemento fixo f corresponde ao único 1-ciclo em P_s); a segunda condição é dada pela Propriedade 2; e, finalmente, as Propriedades 5 e 7 garantem a terceira condição. Logo, P_s é canônica. □

Lema 4 *Se G é um grafo de permutação redutível canônico, então G é uma marca d'água produzida pelo algoritmo de Chroni e Nikolopoulos [32] para algum identificador inteiro ω.*

Prova: Seja G um grafo de permutação redutível canônico, e T sua árvore representativa. Pelo Lema 2, T é unicamente definida pelo percurso em pré-ordem P. Mostramos que P corresponde à permutação auto-inversível P_s gerada pelo algoritmo de codificação de Chroni e Nikolopoulos [32] que computa a marca d'água para algum identificador inteiro ω. Pela definição, $P = (s_i)$, $i = 1, ..., 2n + 1$, é uma permutação auto-inversível canônica, apresentando um único 1-ciclo f e um número n de 2-ciclos (p, q). Estes 2-ciclos (p, q) definem exatamente uma permutação bitônica $P_b = (b_j)$, $j = 1, ..., 2n + 1$ satisfazendo a Propriedade 1 da marca d'água de Chroni e Nikolopoulos com

(i) $b_{n+1} = f$, e
(ii) para todo $j \in \{1, ..., n\}, b_j = p$ se e somente se $b_{2n+1-j} = q$.

A permutação bitônica P_b pode ser definida como $Z_0||Z_1^R$, sendo Z_0 o prefixo de P_b correspondente ao índices dos 0's no binário extendido B^*, uma subsequência ascendente maximal. Extraindo o binário B, que é formado pelo complemento de

1's da subsequência de B^* com dígitos da $(n+1)$-ésima a $(2n)$-ésima posição em B^*, e tomando B como a representação do inteiro positivo ω, a imagem de ω depois de aplicada a função de codificação de Chroni e Nikolopoulos é isomorfa a G. $\square$

Vamos para a última definição antes de darmos a caracterização apropriada dos grafos de permutação redutíveis canônicos. Seja T a árvore representativa de algum grafo de permutação redutível canônico G, e P uma permutação auto-inversível canônica correspondente a um percurso em pré-ordem sem raiz de T. Considere o elemento fixo f de P que é também o elemento fixo (ou vértice) de G e T. Similarmente, os elementos de 2-ciclos de P correspondem a ciclos (ou vértices) de G e T. Um vértice $v \in V(T) \setminus \{2n+2\}$ é considerado *grande* quando $n < v \leq 2n+1$; por outro lado, quando $v \leq n$, v é dito ser *pequeno*. Denotamos por X, Y os subconjuntos de vértices grandes e pequenos em T, respectivamente. Temos que $|X| = n+1$ e $|Y| = n$. Pelo Lema 1, $f \in X$. Então, definimos $X_c = X \setminus \{f\} = \{x_1, ..., x_n\}$ como o conjunto de ciclos de vértices grandes em T.

Definição 5 *Uma árvore representativa T é uma árvore do Tipo-1 (Figura 3.5(a)) quando*

(i) $n+1, n+2, ..., 2n+1$ *são filhos da raiz* $2n+2$ *em* T, *e*
(ii) $1, 2, ..., n$ *são filhos de* $2n$.

Definição 6 *Uma árvore representativa T é uma árvore do Tipo-2 (Figura 3.5(b)) quando*

(i) $n+1 = x_1 < x_2 < ... < x_\ell = 2n+1$ *são filhos de* $2n+2$, $\forall \ell \in [2, n-1]$;
(ii) $x_i > x_{i+1}$ *e* x_i *é pai de* x_{i+1}, *para todo* $i \in [\ell, n-1]$
(iii) $1, 2, ..., f-n-1$ *são filhos de* x_n;
(iv) $x_i = n+i$, *para* $1 \leq i \leq f-n-1$;

(v) f *é um filho de* x_q*, para algum* $q \in [\ell, n]$ *satisfazendo* $x_{[q+1]} < f$ *quando* $q < n$*; e*

(vi) $N_T^*(f) = \{f - n, f - n + 1, ..., n\}$ *e* $y_i \in N_T^*(f)$ *tem índice* $x_{y_i} - f + 1$ *no percurso em pré-ordem de* $N_T^*[f]$*.*

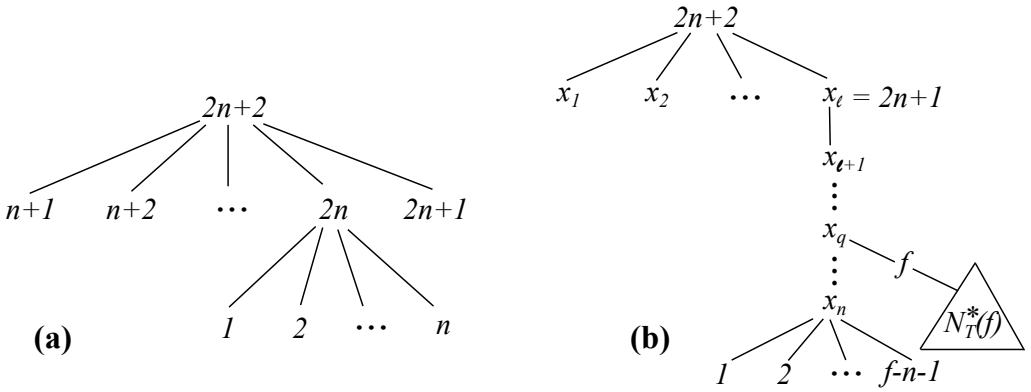


Figura 3.5: (a) Árvore representativa Tipo-1. (b) Árvore representativa Tipo-2.

Lema 5 *Se* y_r *é o vértice mais à direita de uma árvore representativa* T *do Tipo-2 relativa a algum* $f \neq 2n + 1$*, então* y_r *é igual ao número* ℓ *de filhos da raiz* $2n + 2$ *em* T*.*

Prova: Pela definição de árvore representativa Tipo-2, a única não folha filha da raiz $2n + 2$ de T é o vértice mais à direita da raiz, chamado de x_ℓ. Portanto, cada filho x_i de $2n + 2$, para $1 \leq i \leq \ell$, aparece precisamente na i-ésima posição no percurso em pré-ordem sem raiz P de T. Uma vez que, pela definição, P é auto-inversível, e y_r é o $(2n+1)$-ésimo (o último) elemento de P, segue que y_r deve ser igual ao índice de $2n + 1 = x_\ell$ em P, que é, $y_r = \ell$.□

O seguinte teorema caracteriza os grafos de permutação redutíveis canônicos em termos das definições anteriores. Tal caracterização é fundamental para os próximos resultados, uma vez que suas condições simples podem ser verificadas em

tempo linear e abrem espaço para os algoritmos de decodificação e recuperação de ataques que serão apresentados a seguir.

Teorema 1 *Um digrafo G é um grafo de permutação redutível canônico se e somente se G é um grafo de fluxo redutível auto-rotulável e*

(i) o elemento fixo de G é $2n+1$ e G tem uma árvore representativa do Tipo-1; ou

(ii) o elemento fixo de G está entre $[n+2, 2n]$ e G tem uma árvore representativa Tipo-2.

Prova: Seja G um grafo de permutação redutível canônico e T sua árvore representativa. Pela definição, G é um grafo de fluxo redutível auto-rotulável. Seja $P = s_1, ..., s_{2n+1}$ o percurso em pré-ordem sem raiz de T, que é uma permutação auto-inversível canônica, também pela definição. Isto significa, entre outras coisas, que P tem um único elemento fixo f, e que $P' = s_1, ..., s_{n+1}$ é uma subsequência bitônica de P. Como T é descendente, segue que o prefixo A de P' constitui sua subsequência ascendente maximal compreendendo somente vértices que são filhos da raiz $2n + 2$, sendo que o elemento $2n + 1$ é certamente o mais à direita.

Primeiro, considere $f = 2n + 1$. Uma vez que f constitui um 1-ciclo de P, f deve ocupar a posição mais à direita, a $(2n + 1)$-ésima posição em P, sendo f uma folha de T. Além disso, pela Propriedade 4 dos grafos de permutação redutíveis canônicos, segue que P' consiste dos elementos $n+1, n+2, ..., 2n, 1$, por isso, $A = n+1, n+2, ..., 2n$, e estes vértices são, portanto, filhos de $2n+2$ em T. Agora, novamente pela Propriedade 4, os elementos $1, ..., n$ aparecem, nesta ordem à direita de A em P. Considerando que P é um percurso em pré-ordem e uma árvore representativa satisfaz a propriedade max-heap, concluímos que os vértices $1, ..., n$ somente podem ser filhos de $2n$, isto é, T é uma árvore representativa do Tipo-1, como necessário.

Agora, suponha $f < 2n+1$. Pelo Lema 1, segue que $f \in [n+2, 2n]$. Sabemos que os filhos de $2n+2$ são os vértices de A. Seja D o subconjunto formado pelos vértices restantes de P'. Claramente, os vértices de D aparecem em ordem decrescente. Uma vez que T satisfaz a propriedade max-heap e P é um percurso em pré-ordem de T, segue que o vértice grande de D é um filho de $2n+1$, e conseguentemente cada vértice em D é pai em T de vértices que estão à sua esquerda ao longo de D. Novamente, por T satisfazer a propriedade max-heap, $f \in [n+2, 2n]$ é filho do vértice pequeno $x_q \in D \cup \{2n+1\}$ satisfazendo $x_q > f$. Vamos examinar novamente a subsequência ascendente A. Sabemos que o primeiro vértice de A é $n+1$. Suponha que os k primeiros vértices de A são $n+1$, n+2, ..., n_k, para algum k. Como P é auto-inversível, segue que os vértices $1, 2, ..., k$ devem ser filhos do último (isto é, menor) vértice de D, e $k = f - n - 1$ pela Propriedade 4. Resta apenas descrever como os demais vértices pequenos, ou seja, $f-n, f-n+1, ..., n$ aparecem em T. Como eles aparecem depois de f em P, podemos concluir que este subconjunto contém exatamente os descendentes $N_T^*(f)$ de f em T. Cada um dos vértices $y \in N_T^*(f)$ constituem um 2-ciclo com algum vértice x presente na subsequência bitônica P', onde o índice de y em P é exatamente x, e todas as condições que definem uma árvore representativa do Tipo-2 são verificadas.

Caso contrário, suponha $f > 2n+1$, seja G um grafo de fluxo redutível auto-rotulável. Inicialmente, suponha que (i) se aplica e seja T a árvore representativa do Tipo-1 correspondente. Então o percurso em pré-ordem sem raiz P de T é $n+1, n+2, ..., 2n, 1, 2, ..., n, 2n+1$. Referindo-se P como uma permutação de $\{1, ..., 2n+1\}$, observamos que $2n+1$ é o único elemento fixo na permutação; para $1 \leq i \leq n$, cada elemento $n+i$ de P tem índice i, enquanto i tem índice $n+1$, e $n+1, n+2, ..., 2n, 1$ forma uma subsequência bitônica de P. Conseguentemente, P é uma permutação auto-inversível canônica, e G é um grafo de permutação redutível canônico.

Finalmente, suponha que (ii) se aplica. Seja T a árvore representativa do Tipo-2 correspondente para algum $f \in [n+2, 2n]$. O percurso em pré-ordem sem raiz P de T consiste de

$$x_1, ..., x_\ell, x_{\ell+1}, ..., x_q, x_{q+1}, ..., x_n, 1, 2, ..., f-n-1, f, P(N_T^*(f)),$$

em que $x_1 = n+1$; $x_\ell = 2n+1$; $x_i = n+1$ para $1 \leq i \leq f-n-1$; $x_1, x_2, ..., x_n, 1$ é uma subsequência bitônica de P; e $P(N_T^*(f))$ denota o percurso em pré-ordem dos vértices de $N_T^*(f)$, em que cada $y_i \in N_T^*(f)$ tem índice $x_{y_i} - f + 1$. Observe que, para $1 \leq i \leq f-n-1$, $(n+i, i)$ constitui um 2-ciclo em P. Além disso, para $f-n \leq i \leq n$, um vértice x_i forma um 2-ciclo com um elemento $y_i \in N_T^*(f)$. Todas as condições são verificadas, logo P é uma permutação auto-inversível canônica e G é um grafo de permutação redutível canônico. □

Corolário 1 *O reconhecimento dos grafos de permutação redutíveis canônicos pode ser realizado em tempo linear.*

Prova: Direto do Teorema 1 e pela definição de grafo de fluxo redutível auto-rotulável, árvores representativas Tipo-1 e Tipo-2, todas as condições podem ser facilmente verificadas em tempo linear. □

3.3 Novo algoritmo de decodificação

Com a caracterização da classe dos grafos de permutação redutíveis canônicos, podemos agora formular um novo algoritmo de decodificação mais simples que o apresentado por Chroni e Nikoloupolos [33]. Os passos da nova decodificação são apresentados no Algoritmo 1.

Algoritmo 1: decodificar(G)

Entrada: marca d'água G com $2n + 3$ vértices
Saída: identificador ω codificado por G

1. seja H o único caminho hamiltoniano de G.
2. rotule os vérices de G de modo que seus rótulos em H sejam $2n + 2$, $2n + 1$, ..., 0.
3. seja $A = N_G^-(2n + 2)$, com $|A| = \ell$, e seja $x_1, ..., x_\ell - 1 \in A$ a sequência ascendente de vértices de $A \setminus \{2n + 1\}$.
4. retorne $\omega = \sum_{i=1}^{\ell-1} 2^{2n-x_i}$.

Teorema 2 *Seja ω um identificador e G a marca d'água correspondente a ω. Seja $A' = x_1, ..., x_{l-1}$ a sequência ascendente de filhos de $2n + 2$ na árvore representativa T de G, que é diferente de $2n + 1$. Então*

$$\omega = \sum_{x_i \in A'} 2^{2n-x_i}.$$

Prova: Os filhos de $2n + 2$ em T são os vértices x_i de G que são cauda de alguma aresta de árvore de G apontando para $2n + 2$. Pela Propriedade 8 dos grafos de permutação redutíveis canônicos, tais vértices $x_i \neq 2n + 1$ são precisamente aqueles que satisfazem $x_i = n + z_i$, onde z_i é o índice de um dígito 1 na representação binária B de ω. A soma de ω pode agora ser facilmente verificada, porque o valor relativo a posição z_i de um dígito 1 é $2^{n-z_i} = 2^{n-(x_i-n)} = 2^{2n-x_i}$. □

Como consequência do Teorema 2, sempre que a marca d'água de entrada não for alterada, o Algoritmo 1 é capaz de recuperar o identificador codificado de uma forma muito simples. Note que, neste caso, não é necessário obter a árvore representativa da marca d'água, uma vez que o conjunto A pode ser determinado como $A = N_G^-(2n + 2)$.

Teorema 3 *O Algoritmo 1 recupera corretamente um identificador codificado como*

um grafo de permutação redutível canônico G em tempo linear.

Prova: Seja G um grafo de permutação redutível canônico com $2n + 3$ vértices codificando um identificador ω. O conjunto A obtido no passo 3 do Algoritmo 1, corresponde aos filhos de $2n+2$ na árvore representativa T de G, tal que, os vértices x_i de G, para $i = 1, ..., \ell$, que são cauda de alguma aresta de árvore de G apontando para $2n+2$. Pela Propriedade 8 dos grafos de permutação redutíveis canônicos, tais vértices x_i, exceto o vértice $2n + 1$, satisfazem $x_i = n + z_i$, onde z_i é o índice de um dígito 1 na representação binária B de ω. A soma de ω pode agora ser facilmente verificada, porque o valor relativo a posição z_i de um dígito 1 é $2^{n-z_i} = 2^{n-(x_i-n)} = 2^{2n-x_i}$.

Para o primeiro passo, seja H inicialmente o único vértice $v_0 \in V(G)$ com grau de saída zero. Atualize H concatenando à sua esquerda o único vértice v_1 que é vizinho de entrada de v_0 em G. Agora, enquanto $V(H) \neq V(G)$, continue concatenando à esquerda de H o único vértice v_h, com $h = |H|$, que é vizinho de entrada de v_{h-1} em $G - \{v_{h-2}, v_{h-3}, ..., v_0\}$. Todo processo pode ser facilmente executado em tempo linear $O(|V(G)| + |E(G)|) = O(n)$ usando listas de adjacência para os vizinhos de entrada e saída de cada vértice.

No terceiro passo do algoritmo, a ordem ascendente pode ser obtida em tempo linear verificando na matriz de adjacência se v é um vizinho de entrada de $2n + 2$, para cada $v \in \{n + 1, n + 2, ..., 2n\}$. Note que isto não é necessário para obter a árvore representativa de G. Os passos 2 e 4 são computados trivialmente em tempo linear.□

Corolário 2 *Ataques de distorção com modificação de k arestas (inserções/deleções) em um grafo de permutação redutível canônico G, com $|V(G)| = 2n + 3$, para*

$n > 2$, podem ser detectados em tempo polinomial, se $k \leq 5$, e recuperados, se $k \leq 2$. Sendo esses limites justos.

Prova: Pelo Teorema 3, sabemos que, para $n > 2$, não há duas marcas d'água G_1, G_2, com $|V(G_1)| = |V(G_2)| = 2n + 3$, tal que

$$|E(G_1) \setminus E(G_2)| = |E(G_2) \setminus E(G_1)| \leq 2,$$

ou seja, sempre é possível recuperar a marca d'água original após a remoção de até duas arestas. Portanto, para $n > 2$, quaisquer dois grafos de permutação redutíveis canônicos G_1, G_2 satisfazem

$$|E(G_1) \setminus E(G_2)| = |E(G_2) \setminus E(G_1)| \geq 3,$$

onde G_1 não pode ser transformado em G_2 por não menos que 6 modificações de arestas. Como a classe dos grafos de permutação redutíveis canônicos pode ser reconhecida diretamente em tempo polinomial pela caracterização dada no Teorema 1, e como qualquer número $k \leq 5$ de modificações de arestas transformam um grafo G da classe em outro grafo G' que não pertence a classe, todos os ataques de distorção com tal magnitude ($k \leq 5$) podem ser detectados. Agora, para $k = 2$, há três possibilidades:

(i) duas arestas foram removidas;
(ii) duas arestas foram inseridas;
(iii) uma arestas foi removida e uma aresta foi inserida.

Se o caso (i) acontecer, isto é, duas arestas forem removidas, o Teorema 3 garante que o grafo original pode ser restaurado totalmente. Se o caso (ii) ou o

caso (iii) acontecer, então um algoritmo simples em que todas os conjuntos possíveis de modificação de arestas são adicionados novamente à marca d'água danificada G' prova que o grafo original G pode ser restaurado em tempo polinomial, uma vez que, como já sabemos, exatamente um desses conjuntos transformam G' em um grafo de permutação redutível canônico. O caso $k = 1$ é simples e pode ser resolvido de maneira análoga.

Resta provar que esses limites são justos. Para $n > 3$, seja $\omega_1 = 2^{n-1}$ e $\omega_2 = 2^{n-1} + 1$. É fácil ver que os grafos G_1, G_2 codificam ω_1, ω_2 satisfazendo (i). Adicionalmente, por construção, a permutação auto-inversível associada a ω_1 é $(n+1, 2n+1, 2n, 2n-1, 2n-2, ..., n+3, 1, n+2, n, n-1, ..., 4, 3, 2)$, e a permutação auto-inversível associada a ω_2 é $(n+1, 2n, 2n+1, 2n-1, 2n-2, ..., n+3, 1, n+2, n, n-1, ..., 4, 2, 3)$. Podemos identificar duas transposições entre essas permutações, nomeadas $(2n, 2n+1)$ e $(2, 3)$. Os efeitos de tais transposições são:

- aresta de árvore cuja cauda é $2n$ indo para $2n+1$ em G_1 e para $2n+2$ em G_2;
- aresta de árvore cuja cauda é $2n-1$ indo para $2n$ em G_1 e para $2n+1$ em G_2;
- aresta de árvore cuja cauda é 2 indo para 3 em G_1 e para 4 em G_2.

Como todas as outras arestas de árvore permanecem as mesmas, há exatamente três vértices em que seus pais na árvore representativa são diferentes em G_1 e G_2, nomeados $2n, 2n-1$ e 2.

Observamos que há muitos pares ω_1, ω_2 para cada valor de n. O caso acima é apenas um exemplo. De fato, os resultados computacionais apresentados na Seção 3.6 sugerem que a proporção de grafos de permutação redutíveis canônicos que codificam binários de n bits, que podem ser modificados em outro grafo de per-

mutação redutível canônico do mesmo tamanho substituindo três arestas, cresce com n. □

3.4 Decodificação linear ($k \leq 2$ arestas removidas)

Nesta seção, analisaremos o efeito causado por ataques de distorção a uma marca d'água (isto é, um grafo de permutação redutível canônico) G em que $k \leq 2$ arestas foram removidas. Note que o único caminho hamiltoniano H de G pode ser destruído pelo ataque. O conhecimento de H é crucial para determinar os rótulos dos vértices (valores de $2n+2$ a 0 ao longo de H). Nossa primeira tarefa é determinar se alguma aresta do caminho de G foi removida e então recuperar H e os rótulos dos vértices apropriadamente.

3.4.1 Recuperação do caminho hamiltoniano

O algoritmo dado em pseudocódigo como Algoritmo 2, recupera o único caminho hamiltoniano H de uma (possivelmente danificada) marca d'água G', que é um grafo isomorfo a um grafo de permutação redutível canônico G menos $k \leq 2$ arestas. O Algoritmo 2 utiliza duas subrotinas apresentadas separadamente: *Conecta próximo subcaminho* e *Valida rótulos*. O algoritmo em si é simples, basicamente computa candidatos a caminho hamiltoniano para G (possivelmente pela reinserção de algumas arestas) e verifica se a rotulagem imposta aos vértices para cada caminho candidato satisfaz algumas condições. O algoritmo retorna um e somente um candidato que passa no teste.

O procedimento *Conecta próximo subcaminho*, dado como Algoritmo 3, recebe como entrada um grafo G', um caminho Q com $V(Q) \cup V(G) = \emptyset$, e uma

Algoritmo 2: Reconstrução do caminho hamiltoniano(G')

Entrada: marca d'água danificada G' com $2n+3$ vértices e até 2 arestas perdidas
Saída: o único caminho hamiltoniano H da marca d'água original G

1. seja V_0 o conjunto de todos os vértices com grau de saída zero em G'.
2. seja H o conjunto de todos os caminhos hamiltonianos candidatos obtidos por uma chamada à *Conecta próximo subcaminho*$(G', V_0, \emptyset)$
3. **para cada** caminho hamiltoniano candidato $H \in \mathcal{H}$ **faça**
   se *Valida rótulos*(G', H) **então**
   retorne H

lista de saída H, onde caminhos hamiltonianos candidatos (restaurados) de G' serão colocados depois de serem concatenados (à esquerda) de uma cópia de Q. O algoritmo inicia determinando um conjunto S de subcaminhos cabeça. Este conjunto compreende todo vértice $s \in V(G')$ satisfazendo $d^+_{G'}(s) = 0$, caso $Q = \emptyset$, ou $d^+_{G'}(s) \leq 1$, caso contrário. Em seguida, calcula a coleção de todos os caminhos invertidos não bifurcados de G' alcançando S (ou S-bups). Um S-bup de G' é um caminho $v_j, v_{j-1}, ..., v_1$ tal que

- $v_1 = s$, para algum $s \in S$;
- $(v_k, v_{k-1}) \in E(G')$, para $2 \leq k \leq j$; e
- o grau de entrada de v_k em $G' - \{v_1, ..., v_{k-1}\}$ satisfaz

$$d^-_{G'-\{v_1,...,v_{k-1}\}}(v_k) = \begin{cases} 0, & se\, 1 \leq k \leq j-1; \\ 1, & se\, k = j. \end{cases}$$

Em outras palavras, iniciando em algum subcaminho cabeça $s = v_1$, o procedimento *Conecta próximo subcaminho* computa um caminho direcionado invertido Q concatenando um vizinho de entrada de v_k a sua esquerda, para $k \geq 1$, quando o grau de entrada de v_k é 1 no grafo induzido por todos os vértices que ainda não foram concatenados ao caminho. Este procedimento é repetido iterativamente até que o grau de entrada de v_j, para algum j, no grafo de entrada seja zero, após adicionar o caminho obtido pela lista de S-bups, ou maior do que um, após descartar o

Algoritmo 3: Conecta próximo subcaminho$(G', Q, \mathcal{H})$

Entrada: grafo G', caminho Q com $V(Q) \cup V(G') = \emptyset$, e uma lista de saída $\mathcal{H}$
Saída: lista atualizada $\mathcal{H}$ contendo todos os $Q'||Q$ onde Q é um caminho hamiltoniano de G' (mais $k \geq 0$ arestas extras) terminando em um vértice com grau $d \leq 1$ ($d = 0$, se Q é vazio)

1. seja $S \leftarrow \{s \in V(G') : d^+_{G'}(s) = 0\}$.
   se $Q \neq \emptyset$ **então** $S \leftarrow \cap\{s \in V(G) : d^+_{G'}(s) = 1\}$
2. **para cada** $s \in S$ **faça**
   $v \leftarrow s$
   $Q' \leftarrow s$
   enquanto $|N^-_{G'-(V(Q')-\{v\})}(v)| = 1$ **faça**
   $v \leftarrow$ o único elemento em $N^-_{G'-(V(Q')-\{v\})}(v)$
   $Q' \leftarrow v||Q'$
   se $|Q'| = |V(G')|$ **então**
   $\mathcal{H} \leftarrow, \mathcal{H} \cup \{Q'||Q\}$
   senão se $|N^-_{G'-(V(Q')-\{v\})}(v)| = 0$ **então**
   Conecta próximo subcaminho$(G' - V(Q'), Q'||Q, \mathcal{H})$
   senão descarta Q' // uma bifurcação invertida foi encontrada

caminho atual. A ideia é que uma bifurcação invertida em v_j significa que existem dois vértices, digamos u e w, que ainda não foram incluídos no caminho, com ambos sendo vizinhos de entrada de v_j. Uma vez que ao menos um deles, digamos u, pode ser cauda de uma aresta de caminho para v_j em Q, o outro, w, será cauda de uma aresta de árvore apontando para v_j, o que não é aceito já que w estará a esquerda de v_j no caminho. Seja qual for o caso, o algoritmo começa novamente com um outro subcaminho com cabeça $s \in S$ até que todos tenham sido considerados e a lista de S-bups estiver totalmente preenchida. Finalmente, o procedimento adiciona cada S-bup Q', um de cada vez, à esquerda de Q (pela adição de uma possível aresta de caminho $e \notin E(G')$ a partir do vértice mais à direita em Q' para o vértice mais à esquerda em Q) e executa uma entre as duas ações possíveis:

- se $V(Q') = V(G')$ (isto é, se Q' é um caminho hamiltoniano de G'), então adicionamos o novo caminho $Q'||Q$ na lista de saída $\mathcal{H}$;

- caso contrário, fazemos uma chamada recursiva a *Conecta próximo subcaminho* com os parâmetros $G' - V(Q')$, $Q'||Q$, e a lista de saída $\mathcal{H}$.

Quando todos os S-bups tiverem sido considerados, a lista $\mathcal{H}$ é retornada.

Se H é um caminho, então indicamos o j-ésimo elemento de H(da direita para esquerda, iniciando por $j = 0$) por $H[j]$.

Algoritmo 4: Valida rótulos(G', H)

Entrada: um grafo G', com $|V(G')| = 2n + 3$, e um caminho hamiltoniano candidato H, com $|E(H) \setminus E(G')| \leq 2$
Saída: *Verdadeiro*, se os rótulos de $V(G')$ atribuídos à H são válidos; *Falso*, caso contrário

1. rotule os vértices de G' de maneira que $H = 2n + 2, 2n + 1, ..., 0$.
   Seja $H^* \leftarrow E(H) \setminus E(G')$, e insira H^* em G' obtendo G''.
   Seja F a floresta obtida a partir de G'' pela remoção de toda aresta (de caminho) em H, além do vértice isolado 0, e *arestas_faltantes* $\leftarrow 0$.
2. **para cada** $v \in \{n + 1, 2n + 1\}$
   se $(v, 2n + 2) \notin E(G'')$ **então**
   se $d^+_{G''}(v) = 2$ **então retorne** *Falso*
   $E(G'') \leftarrow E(G'') \cup \{(v, 2n + 2)\}$; *arestas_faltantes* $+= 1$
   para cada $v \in \{1, ..., 2n + 1\}$
   se $d^+_{G''}(v) < 2$ então *arestas_faltantes* $+= 1$ **então**
   se *arestas_faltantes* $> 2 - |H^*|$ **então retorne** *Falso*
3. **se** os vértices 1 e n são vizinhos em F então
   Seja r o vértice mais à direita no percurso em pré-ordem de F
   se $r > d^-_{G''}(2n + 2)+$ *arestas_faltantes* **então retorne** *Falso*
   seja x o comprimento do único caminho de 1 até $2n + 2$ em F
   se $r + x - 2 \neq n$ **então retorne** *Falso*
4. **retorne** *Verdadeiro*

O segundo procedimento chamado no Algoritmo 2, nomeado *Valida rótulos*, é mostrado em pseudocódigo como Algoritmo 4. Ele toma como parâmetro uma marca d'água G' (com até duas arestas removidas) e um caminho hamiltoniano H. Primeiro, determina o conjunto $H^* = E(H) \setminus E(G')$ com $k \leq 2$ arestas de caminho

que podem ser necessárias para compor H. Então verifica se é possível obter um grafo de permutação redutível canônico G válido depois da inserção de H^* e algum conjunto de $2-k$ arestas de árvore em G'. Neste teste é necessário verificar as seguintes condições, onde T corresponde à árvore representativa de G:

(1) os vértices $H[2n+1]$ e $H[n+1]$ são vizinhos de entrada de $H[2n+2]$ em G;

(2) o grau de saída dos vértices $H[2n+1], ..., H[1]$ deve ser 2 em G e o grau de saída de $H[2n+2]$ deve ser 1;

(3) o número de arestas de árvore que deveriam ser inseridas em G para atender as duas condições anteriores não pode ultrapassar $2-|H^*|$; e finalmente,

(4) se os vértices $H[1]$ e $H[n]$ não são irmãos em T, então o vértice $H[1]$ deve ser filho do n-ésimo descendente da raiz $H[2n+2]$ que é um vértice grande (isto é, o n-ésimo descendente da raiz, contando da direita para a esquerda, de modo que seus índices em H são maiores ou iguais a $n+1$); caso contrário, o índice em H do vértice mais à direita no percurso em pré-ordem de T deve corresponder ao número de filhos de $2n+2$ em T.

A Condição (1) aparece nas definições de árvores representativas de Tipo-1 e Tipo-2 (Definições 5 e 6), que estão diretamente ligadas ao Teorema 1. A Condição (2) ocorre devido à Definição 4 e à segunda propriedade na definição de grafo de fluxo redutível auto-rotulável (Definição 1). A Condição (3) vem do fato que 2 arestas foram removidas de G, e $|H^*|$ arestas precisam ser (re-)inseridas. Finalmente, a Condição (4) é devido à propriedade (iii) na definição de árvore representativa Tipo-2 e do Lema 5. O Algoritmo 4 certamente se aplica à definição de árvore representativa de Tipo-2, que é o caso onde os vértices $H[1]$ e $H[n]$ não são irmãos na árvore representativa, pela definição. Observe que, se H é de fato o único caminho hamiltoniano no grafo de permutação redutível canônico G, então para todo $v \in V(G)$, o rótulo canônico v satisfaz $v = H[v]$.

Teorema 4 *O Algoritmo 2 recupera corretamente o original e único caminho hamiltoniano de um grafo de permutação redutível canônico com $2n + 3$ vértices do qual $k \leq 2$ arestas foram removidas. O algoritmo é executado em tempo $O(n)$.*

Prova: Seja $\mathcal{G}_k$ o conjunto de todos os grafos de permutação redutíveis canônicos com $k \leq 2$ arestas perdidas. Quando um elemento G' de $\mathcal{G}_k$ é entrada de *Conecta próximo subcaminho*$(G', \emptyset, \mathcal{H})$, a saída é claramente um caminho hamiltoniano de algum grafo G tal que $V(G) = V(G')$ e $E(G) \backslash E(G') \leq k$. Então, quando um grafo de permutação redutível canônico G menos duas arestas é passado para o Algoritmo 2, o caminho H retornado é um caminho hamiltoniano para algum elemento de $\mathcal{G}_2$. Afirmamos que esse grafo não pode ser outro senão G.

Seja $\hat{H} = 2n+2, 2n+1, ..., 0$ o único caminho hamiltoniano de G. Dividimos a prova em três casos:

(i) as arestas removidas são arestas de árvore de G;
(ii) as arestas removidas são arestas de caminho de G;
(iii) as arestas removidas são uma de árvore e outra de caminho de G.

A Figura 3.6 mostra os cenários possíveis para o caminho hamiltoniano de uma marca d'água danificada G'. As linhas pontilhadas indicam as arestas removidas. Quadrados, círculos sólidos e círculos vazados representam vértices cujo grau de saída em G' são, respectivamente, 0, 1 e 2. Três círculos vazados juntos seguidos de uma aresta (com til no meio) indicam um subcaminho de zero ou mais arestas. Cada retângulo grande agrupa um subcaminho não danificado maximal de H, que corresponde a um caminho não bifurcado maximal para $s \in V(G)$, isto é, um $\{s\}$-bup.

Quando duas arestas de árvore são removidas. O caso (i) é o mais fácil e

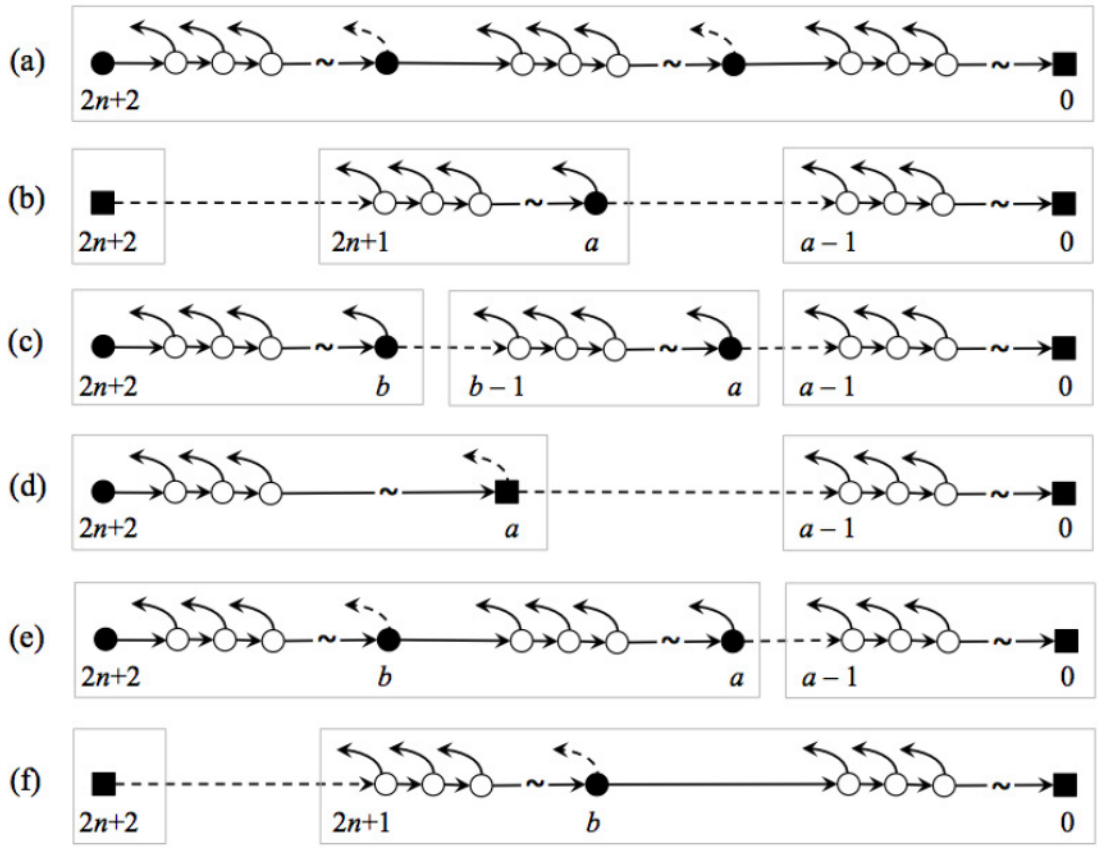


Figura 3.6: Cenários possíveis para o caminho hamiltoniano de uma marca d'água danificada G' após a remoção de até duas arestas.

é ilustrado na Figura 3.6(a). Se somente arestas de árvore foram removidas, então o caminho hamiltoniano de G não foi danificado. Iniciando pelo único vértice com grau de saída zero em G', nomeado vértice 0, o algoritmo *Conecta próximo subcaminho*$(G', \emptyset, \mathcal{H})$ fornece $\hat{H}$ como saída sem fazer nenhuma chamada recursiva. Uma vez que $\hat{H}$ obviamente produz uma rotulação correta dos vértices de G, é validado com sucesso pelo algoritmo *Valida rótulos*$(G', \hat{H})$ e retornado pelo algoritmo.

Quando duas arestas de caminho são removidas. Suponha agora que é o caso (ii). Uma vez que nenhuma aresta de árvore foi removida, o único vértice com grau de saída igual a zero em G' é o vértice 0, a menos que a aresta de caminho

cujo a cauda é $2n+2$ foi uma das arestas removidas (temos que $2n+2$ tem grau de saída 1 em G). Vamos analisar dois casos, de acordo com a remoção ou não de $(2n+2, 2n+1)$ de $E(G)$.

Para o primeiro subcaso, suponha a remoção das arestas de caminho $(2n+2, 2n+1)$ e $(a, a-1)$ para algum $a \geq 1$, como na Figura 3.6(b). Embora o vértice $2n+2$ tenha grau de saída zero, seu grau de entrada é maior que 1 em $G' \setminus \emptyset = G'$, e, portanto, nenhum $\{2n+2\}$-bup é produzido pela chamada a *Conecta próximo subcaminho*. Logo, o único caminho parcial produzido pelo algoritmo, iniciando no vértice 0 de maneira invertida é $Q' = a-1, a-2, ..., 0$. Como agora $a-1$ não tem vizinho de entrada em $G - V(Q')$, ele continua a recursão para encontrar possíveis extensões para Q'. Os únicos vértices com grau de saída 0 ou 1 em $G' \setminus V(Q')$ agora são $2n+2$ e a. De qualquer forma, qualquer $\{2n+2\}$-bup Q'' que pode ser encontrado constituirá um caminho $H = Q''||Q$ que, necessariamente, não poderá ser validado. Isto se deve ao fato de que $H[2n+2]$ será um vértice diferente de $\hat{H}[2n+2] = 2n+2$, um vizinho de saída de $n+1$ pelo Teorema 1 — e a primeira condição testada por *Valida rótulos*(G', H) não pode ser atendida. Por isso, o caminho parcial Q' só pode ser extendido por um $\{a\}$-bup, que só pode ser $2n+1, ..., a$ e o vértice $2n+2$ remanescente, que será concatenado durante a próxima chamada recursiva, completando o caminho hamiltoniano $2n+2, 2n+2, ..., a, a-1, ..., 0 = \hat{H}$ como desejado.

O segundo subcaso é aquele em que a aresta do caminho $(2n+2, 2n+1)$ não foi removida. Suponha que as arestas do caminho $(a, a-1)$ e $(b, b-1)$ foram removidas, com $a < b$, como ilustrado na Figura 3.6(b). Primeiro, o subcaminho mais à direita localizado pelo algoritmo só pode ser $\{0\}$-bup, nomeado $Q' = a-1, a-2, ...0$. Agora há três vértices cujo grau é menor ou igual a um: $2n+2, b$ e a.

Quando o algoritmo considerar $\{2n+2\}$-bup durante a chamada recursiva

de *Conecta próximo subcaminho*$(G' - V(Q'), Q'||\emptyset, \mathcal{H})$, qualquer que seja o caminho hamiltoniano candidato H será necessariamente descartado. Adicionalmente, se $n+1 \geq a$, então nenhum bup é produzido porque o grau de entrada de $2n+2$ em $G' - V(Q')$ é ao menos 2 pela existência das arestas de árvore $(n+1, 2n+2)$ e $(2n+1, 2n+2)$; e se $n+1 < a$, então $H[n+1]$ é o vértice $\hat{H}[n+1] = n+1$ e, como seu grau de saída em G' já é 2, as Condições (1) e (2) verificadas por *Valida rótulos*(G', H) não são atendidas.

Quando o algoritmo considera um $\{b\}$-bup, seja qual for o caminho hamiltoniano candidato H ele também será descartado. De fato, como o subcaminho $2n+2, 2n+1, ..., b$ de $\hat{H}$ está intacto, o vértice $2n+2$ será traduzido em um $\{b\}$-bup antes de v, para todo $b-1 \geq v \geq a$, por isso $H[2n+2] \neq 2n+2$. Além disso, como a aresta de árvore cuja cauda é $b-1$, digamos $(b-1, w)$, não foi removida, e $w \leq b$, o único valor possível para w é $2n+2$, caso contrário, haveria um vértice $z \in \{2n+1, 2n, ..., b\}$ com grau de entrada maior que 1 no subgrafo de G' induzido por z e pelos vértices a esquerda de z em H. Este fato é uma contradição, já que tal caminho teria sido rejeitado na última linha do Algoritmo 3. Assim, o vértice $b-1$ é um vizinho de entrada de $2n+2$ que não foi adicionado no caminho antes de $2n+2$ ser adicionado. Caso uma bifurcação invertida não tenha surgido, então só é possível que $b-1$ seja o vértice à esquerda de $2n+2$ em H. Repetindo o mesmo argumento — com base no fato de que a aresta de árvore cuja cauda é v não foi removida — para todo $b-2 \geq v \geq a$, podemos inferir que somente um caminho hamiltoniano candidato possível é produzido pela concatenação de um $\{b\}$-bup à esquerda de Q', que é $H = a, a+1, ..., b-1, 2n+2, 2n+1, ..., b, a-1, a-2, ..., 0$.

Agora, a condição (1) em *Valida rótulos* impõe que $H[n+1] > a$ é cauda de uma aresta de árvore que aponta para $H[2n+2] = a$. No entanto, como $H[n+1] > 0$, tal aresta não pode ser uma aresta de árvore do grafo original G, já que deve ser uma aresta de caminho. Uma vez que a única aresta de caminho com cabeça a em G é

$a+1$, segue que $H[n+1] = a+1$. E aqui teremos uma contradição, uma vez que $a+1$ é o segundo vértice, da esquerda para a direita, em H (isto é, $H[2n+1] = n+1$), a menos que $a = b-1$. No entanto, se $a = b-1$, então $H[n+1] = b$ e a existência da aresta $(b, a) = (H[n+1], H[2n+2])$ é necessariamente encontrada pela condição (1) do algoritmo de validação. Mas $(b, a) = (b, b-1)$ é uma das arestas removidas, por isso, deve ser reinserida. A condição (2), por sua vez, requer que uma aresta de saída seja adicionada com destino em $2n+2$ (cujo grau é 1 e cujo índice i em H satisfaz $2n+1 \geq i \geq 1$). Juntamente com a provável aresta de caminho $(b, a-1)$, que foi necessária para concatenar o $\{b\}$-bup à esquerda de Q', temos um total de três novas arestas, o que viola a condição (3).

Finalmente, quando o algoritmo considerar o $\{a\}$-bups, que produz necessariamente o subcaminho $Q'' = b-1, ..., a$, que é concatenado com Q', e, como $\{2n+2\}$-bups não pode produzir um prefixo válido para $Q''||Q'$, a última recursão só pode produzir o $\{b\}$-bup $2n+2, ..., b$, que completa a reconstrução de $\hat{H}$.

Quando uma aresta de árvore e uma aresta de caminho são removidas. Agora vamos analisar o caso (iii), onde uma aresta do caminho hamiltoniano e uma aresta de árvore foram removidas. Vamos considerar três subcasos separadamente. No primeiro, as arestas de árvore e do caminho que foram removidas possuem a mesma cauda. No segundo caso, a cauda das arestas removidas são distintas. O terceiro caso é um caso especial do segundo caso, onde a cauda da aresta removida é $2n+2$.

Para o primeiro caso, ilustrado na Figura 3.6(d), digamos que ambas as arestas removidas possuem cauda em $a \in V(G')$. Neste caso, o vértice a possui grau de saída zero, igual ao vértice 0. Qualquer tentativa de construir um caminho hamiltoniano H com um sufixo $\{a\}$-bup retorna falha. Uma vez que o vértice $2n+2$ que será apresentado no caminho H antes do vértice 0, e como a será o vértice mais à direita em H (isto é, $H[0] = a$), uma aresta de árvore que parte de $2n+2$ satisfaz

a condição (2) de *Valida rótulos(G',H)*. Mas o vértice 0 aparece com índice $i > 0$ em H, e, portanto, uma aresta de caminho possível deve ser inserida com 0 sendo sua cauda. Se o índice de 0 não é $2n + 2$, então uma aresta de árvore que deixa 0 também é necessária. Se, por outro lado, o índice de 0 é $2n+2$, então, entre as duas arestas de árvore que alcançam $H[2n + 2] = 0$ que são requeridas pela condição (1) do procedimento de validação, ao menos uma delas foi perdida. Em ambos os casos, a condição (3) é violada.

O segundo caso é apresentado na Figura 3.6(e), onde uma aresta de caminho $(a, a - 1)$, com $1 < a \leq 2n + 1$, e uma aresta de árvore (b, v), com $v > b$, foram removidas. O procedimento *Conecta próximo subcaminho* começa tomando um caminho invertido não bifurcado Q' com cabeça igual a 0, o único vértice com grau zero em G'. O vértice mais à direita do $\{0\}$-bup é o vértice $a - 1$, o primeiro vértice cujo grau de entrada é zero no subgrafo G' induzido pelos vértices que não estão em Q', e onde $Q' = a - 1, a - 2, ..., 0$. Agora três vértices possuem grau de saída maior ou igual a um: $2n + 2, b$ e a.

Quando o algoritmo considera $2n + 2$ como uma possível continuação do caminho invertido que está sendo construído, o índice de $2n + 2$ em H será a. Pelo Teorema 1, o vértice $2n+1$ é sempre filho da raiz $2n+2$ na árvore representativa T de um grafo de permutação redutível canônico G, e pela Propriedade 8, o número de filhos $v \leq 2n$ de $2n+2$ em T corresponde ao número n_1 de dígitos 1 na representação binária B do identificador ω codificado como G. Como consequência, o grau de entrada de $2n + 2$ em G é $n_1 + 1$. Agora vamos tratar duas situações distintas. Na primeira, $a \leq n + 1$, e na segunda $a > n + 1$.

Se $a \leq n+1$, então o grau de entrada de $2n+2$ em $G' - V(Q')$ é o mesmo que em G' (isto é, n_1), uma vez que todos os vizinhos de entrada de $2n + 2$ pertencem a $\{n+1, ..., 2n+1\}$ pelo Teorema 1. Como, ao longo das arestas de caminho $(a, a-1)$,

somente uma aresta de árvore foi removida de G para obter G', o grau de entrada de $2n+2$ em G' é ao menos $n_1+1-1=n_1$. Como consequência, uma bifurcação invertida seria encontrada em $2n+2$, a menos que $n_1=1$ e a cauda b da aresta de árvore removida seja um vizinho de entrada de $2n+2$, que, neste caso, são $n+1$ e $2n+1$. Se $b=n+1$, então a aresta de árvore $e=(2n+1,2n+2)$ está intacta, e o único lugar possível para inserir o vértice $2n+1$ em H é ocupando a posição imediatamente à esquerda de $2n+2$, então e funciona como uma aresta de caminho de H. Assumindo que não ocorre uma bifurcação invertida em $2n+2$ (o que teria causado a exclusão do caminho H), a única aresta de árvore possível que parte de $2n$ é $(2n,2n+1)$. Portanto, $2n$ deve ser colocado à esquerda de $2n+1$ em H. Analogamente, assumindo que não ocorreu uma bifurcação invertida em $2n+1$, a única aresta de árvore possível que deixa $2n-1$ é $(2n-1,2n)$, e assim por diante. Continuando esse raciocínio até que finalmente a seja concatenado na primeira posição de H, obtemos $H=a,a+1,...,2n+2,a-1,a-2,...,0$.

Agora, a condição (1) do procedimento de validação requer que $H[n+1]$ e $H[2n+1]$ sejam vizinhos de entrada de $H[2n+2]=a$. Contudo, este requisito e a condição (2) não podem ser satisfeitos sem violar a condição (3), porque, uma vez que os vértices $H[n+1]$ e $H[2n+1]$ não estão em Q', eles são certamente maiores do que a, mas há somente um vértice em G que é maior do que a e é o vizinho de entrada de a, isto é, $a+1$. Assim, é necessário adicionar uma aresta de árvore extra, mas uma aresta de árvore também é solicitada pela condição (2) — a aresta de árvore que deixa b — e a plausível aresta de caminho $(2n+2,a-1)$ já havia sido inserida, o que viola a condição (3). Ficamos com a possibilidade de que a cauda da aresta de árvore removida seja $b=2n+1$. Neste caso, a aresta de árvore $(n+1,2n+2)$ está intacta, e o vértice imediatamente a esquerda de $2n+2$ em H deve ser $n+1$.

Uma vez que a aresta de caminho $(n+2,n+1)$ não foi removida por hipótese,

o vértice $n+2$ deve estar imediatamente a esquerda de $n+1$ em H, e uma vez que a aresta de caminho $(n+3, n+2)$ não foi removida, o vértice $n+3$ deve aparecer imediatamente à esquerda de $n+2$, e assim por diante, até que $b = 2n+1$ é concatenado na primeira posição de H, obtendo-se $H = 2n+1, 2n, ..., a, 2n+2, a-1, a-2, ..., 0$. Para satisfazer a condição (1) da validação, o vértice $H[n+1]$ deve ser um vizinho de entrada de $H[2n+2] = 2n+1$. Mas, como $n_1 = 1$ (ω é potência de 2), a árvore representativa Tipo-2 de G possui somente dois filhos, e seguindo o item (iii) da Definição 6 podemos assumir que $2n+1$ tem somente um filho, e este filho não é $n+1$, pelo item (i) da mesma definição. Assim, a aresta de árvore $(H[n+1], 2n+1)$ deve ser adicionada para satisfazer a condição (1) de *Valida rótulos(G',H)*, e os únicos vértices com grau de saída 1 em G' eram b, que é o $2n+1$, $2n+2$, que já foi adicionada uma possível aresta de caminho conectando-o à $a-1$, e a.

Como consequência, é possível apenas que $H[n+1] = a$, isto é, a aresta de caminho removida é necessariamente $(n+1, n)$. E aqui a condição (4) do procedimento entra em jogo, reforçando que a raiz $H[2n+2]$ apresenta apenas dois filhos quando ω é uma potência de 2. Uma vez que não é o caso para H ser obtido, como pode ser facilmente verificado, H é descartado. A segunda situação é a em que $a > n+1$. Esta é fácil, já que agora $H[n+1] = n+1$, que é a cauda da aresta de árvore que aponta para $2n+2 \neq H[2n+2]$, e, por conseguinte, as condições (1) e (2) não podem ambas serem satisfeitas, a menos que tal aresta de árvore seja precisamente uma aresta de árvore que foi removida. Mas que corresponderia ao subcaso mostrado na Figura 3.6(d), que já abordamos.

Quando o algoritmo toma b como cabeça do primeiro subcaminho para estender o $\{0\}$-bup Q', todos os subsequentes candidatos a caminho hamiltoniano devem ser descartados por motivos semelhantes. Finalmente, quando se considera a continuação a, todas as condições são atendidas e $\hat{H}$ é retornado.

A terceira — e última — situação possível é aquela mostrada na Figura 5(f), a aresta de caminho $(2n+2, 2n+1)$ e uma aresta de árvore (b, v), com $v > b$, foram removidas, resultando em dois vértices com grau de saída zero: 0 e $2n+2$. Quando o procedimento *Conecta próximo subcaminho(G', ∅, $\mathcal{H}$)* é chamado e escolhe $2n+2$ como o vértice mais à direita de Q', o vértice mais à esquerda de qualquer caminho hamiltoniano H produzido deve ser 0 ou b, os únicos vértices com grau de saída menor que 2 em G' (parte da segunda condição verificada por *Valida rótulos*). Além disso, a raiz da árvore representativa de G deve ter somente dois filhos (que significa $n_1 = 1$, ou de forma equivalente, que o identificador ω codificado como G é uma potência de 2), e b deve ser $2n+2$ ou $n+1$, de modo que não ocorra uma bifurcação invertida no vértice $H[0] = 2n+2$. Se $H[2n+2] = 0$, então a aresta de caminho $(H[2n+2], H[2n+1])$, e ao menos duas arestas de árvore ($(H[2n+1], H[2n+2])$ e $(H[n+1], H[2n+2])$) são necessárias para satisfazer a condição (1) do procedimento de validação, violando a condição (3). Se $H[2n+2] = b = n+1$, então o vértice imediatamente à esquerda de $H[0] = 2n+2$ em H deve ser $H[1] = 2n+1$, e o próximo vértice da direita para a esquerda deve ser $H[2] = 2n$ e assim por diante, assumindo que bifurcações invertidas não ocorrem ao menos até o vértice $H[n] = n+2$. Dessa forma, o $\{2n+2\}$-bup Q' considerado inicialmente pelo algoritmo contêm (não necessariamente de maneira correta, dependendo se houve arestas de árvore apontando para $n+2$ em G') o sufixo $Q' = n+2, n+3, ..., 2n+2$. Agora, não importa qual vértice w ocupa a $(n+1)$-ésima posição (da direita para a esquerda) em H, certamente não era nenhum vizinho de entrada de $H[2n+2] = n+1$, porque $n+1$ não tem vizinho de entrada em uma árvore do Tipo-2 (e em uma árvore Tipo-1 também, para este problema). Se $w \neq 0$, então w tem grau de saída 1, e as condições (1) e (2) do procedimento de validação não podem ser ambos atendidos. Se, por outro lado, $w = 0$, então H é a concatenação de Q' com o prefixo $n+1, n, n-1, ..., 0$, um subcaminho intacto de $\hat{H}$. Neste caso, o vértice $H[2n+1]$ é n, um vértice com grau de saída 2 em G' que não é vizinho de entrada de $H[2n+2] = n+1$, e as condições (1) e (2), novamente, não podem ser satisfeitas.

As verificações pode ser realizadas diretamente. □

3.4.2 Determinando o elemento fixo

Suponha que a marca d'água G sofreu a remoção de duas arestas quaisquer, resultando numa marca d'água G' danificada. Agora vamos reconhecer o elemento fixo da marca d'água original, dado uma marca d'água danificada. Conhecer o elemento fixo de G é crucial para definir regras para identificação de arestas de árvore pedidas e consequentemente a recuperação do identificador original ω codificado como G.

Descreveremos algumas caracterizações que permitem uma definição eficiente do elemento fixo f de G. Seja T a árvore representativa da marca d'água original G. Consideramos o caso onde duas arestas foram removidas de T. Denote por F a floresta obtida a partir de T pela remoção de duas arestas. Primeiro, considere o caso que $f = 2n + 1$.

Teorema 5 *Seja F uma floresta obtida pela remoção de duas arestas da árvore representativa T, onde $n > 2$. Então, $f = 2n + 1$ se e somente se*

(1) o vértice $2n + 1$ é folha de F; e
(2) os n vértices pequenos de G' são filhos de $2n$ em F, com a possível exceção de ao menos 2 deles, que neste caso, são isolados.

Prova: Pelo Teorema 1, sabemos que, quando $f = 2n + 1$, f é o vértice mais à direita de T, por isso uma folha de F, implicando a necessidade da Condição (1). Novamente, pelo Teorema 1, os vértices pequenos de T devem seguir imediatamente

à direita do vértice cíclico de T, nomeado $2n$. Uma vez que duas arestas foram removidas de T, segue que todos os vértices pequenos são filhos de $2n$ em F, com a possível exceção de ao menos dois deles, que se tornam isolados, então a Condição (2) também é necessária.

Por outro lado, suponha que as Condições (1) e (2) são válidas, e assuma que $f \neq 2n+1$. Então, o segundo caso coberto pelo Teorema 1 aplica-se a T. Se $f = 2n$, sabemos para $n > 2$ que o vértice $2n + 1$ tem ao menos 3 filhos em T, tornando impossível para $2n+1$ se tornar uma folha de F pela remoção de somente 2 arestas, o que contradiz a Condição (1). Se, por outro lado, $f < 2n$, então, novamente pelo Teorema 1, o vértice $2n + 1$ não pode ter nenhum filho pequeno, contradição com a Condição (2). Portanto, $f > 2n$, implica que $f = 2n + 1$. □

A seguir, caracterizamos o caso em que $f < 2n + 1$. A Figura 3.7 ajuda a visualizar as três condições do Teorema 6.

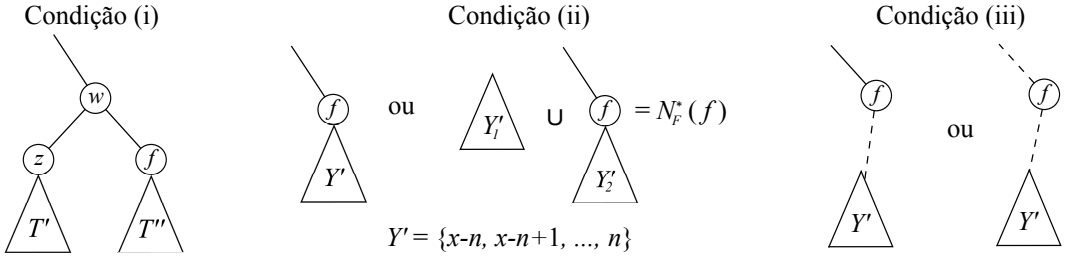


Figura 3.7: Condições (i), (ii) e (iii) do Teorema 6.

Teorema 6 *Seja F uma floresta obtida pela remoção de duas arestas da árvore representativa T de uma marca d'água G, e seja $x \leq 2n$ um vértice grande de T que não é um filho de $2n + 2$. Então, x é o vértice fixo f de G, se e somente se*

(i) o vértice grande x tem um irmão z em F, e $x > z$; ou
(ii) o subconjunto dos vértices pequenos $Y' \subset Y$, $Y' = \{x - n, x - n + 1, ..., n\}$ pode ser particionado em ao menos dois subconjuntos Y'_1, Y'_2, tal que $\emptyset \neq Y'_1 =$

$N_F^+(x)$ *e* Y_2' *é o conjunto dos vértices de uma das árvore que formam* F*; ou, quando não possuir as condições anteriores,*

(iii) o vértice grande x *é o vértice mais à direita de uma das árvores de* F*, enquanto os vértices mais à direita das árvores restantes são todos pequenos.*

Prova: Para satisfazer a Condição (i), seja x um vértice grande de F, z um irmão de x em F e x_q seu pai. Pelo Teorema 1, o único vértice grande de T que não é filho de $2n+2$ e tem algum irmão é precisamente o vértice fixo f. Claramente, a remoção de arestas de T não pode criar um novo vértice com essa propriedade. Além disso, $x_q \in \{x_n, 2n+1\}$ implica que todo irmão y de f é um vértice pequeno, por isso $f > y$. Conseguentemente, $x = f$.

Agora, suponha que a Condição (ii) é válida. Primeiro, assuma que $Y_2' = \emptyset$. Neste caso, $Y' = Y_1' = \{x-n, x-n+1, ..., n\} = N_F^*(x)$. Novamente, de acordo com o Teorema 1, podemos localizar o único vértice f diretamente por esta propriedade, implicando que $x = f$. E mais, quando $Y_2' \neq \emptyset$, podemos novamente selecionar o único vértice f, onde $N_F^*(f) \cup Y_2' = Y'$. Assim, $x = f$ de fato.

Finalmente, assuma que as Condições (i) e (ii) não são válidas. Como (i) não é satisfeita, temos que ou $x_1 \notin \{x_n, 2n+1\}$, e a aresta que parte de x_q para um de seus filhos foi excluída; ou $x_q \in \{x_n, 2n+1\}$, e a aresta (x_q, f) foi excluída. Além disso, como (ii) não é satisfeita, f deve ter um único filho y, e a aresta (f, y) também foi removida. Em seguida, assuma que, em um contexto qualquer, a Condição (iii) é verificada. Por contradição, suponha que o teorema é falso, então $x \neq f$. Como $x \neq 2n+1$ e x não é filho de $2n+2$, segue que deve ser um vértice descendente, baseado no fato de que x é o vértice mais à direita da árvore de F, isso implica que x é uma folha de F, implicando que a aresta (x_q, x) de T foi removida, onde x_q é o pai de x em T. Como a Condição (i) não é satisfeita, pelo menos uma aresta foi removida de T, e como a Condição (ii) não é satisfeita, ao menos uma aresta a

mais foi removida de T. Uma vez que não mais que duas arestas quaisquer foram removidas, concluímos que a suposição é falsa e, portanto, que $x = f$.

Contrariamente, assuma que x é um vértice grande de F, satisfazendo $x = f$. Provamos que a Condição (i) ou a Condição (ii) é válida, caso contrário, a Condição (iii) é satisfeita.

Seja x_q o pai de $x = f$ em T. Se x_q tem ao menos dois filhos em F, então f é maior que seus irmãos pelo Teorema 1 e a Condição (i) é válida. Alternativamente, se f não é uma folha de F, então o conjunto $Y' = \{x - n, x - n + 1, ..., n\}$ ou satisfaz $Y' = N_F^*(f)$, ou pode ser dividido em dois subconjuntos $Y_1' \cup Y_2' = Y'$, onde $Y_1' = N_F^*(f)$ e Y_2' é o conjunto de vértices de uma das árvores de F. Nesta situação, a Condição (ii) é satisfeita. Assuma, em seguida, que nem a Condição (i) nem a Condição (ii) são satisfeitas. Então, o pai x_q de f em T tem ao menos um filho em F, enquanto f não tem filho. Posteriormente, isso implica que f é o vértice mais à direita da árvore de F que o contém. Como $x_q \neq f$, sabemos que não mais que duas arestas foram removidas de T, por isso não há vértices que podem ser folha de F e sejam maiores que f. Consequentemente, a Condição (iii) é satisfeita e isso completa a prova. □

Os teoremas anteriores levam ao Algoritmo 5 que encontra o vértice fixo de uma marca d'água G, no qual a entrada é a floresta F, obtida a partir da árvore representativa T de G após a remoção de duas arestas. Primeiro, o algoritmo verifica se $f = 2n+1$. Pelo Teorema 5, basta verificar se $2n+1$ é uma folha de F e se todos os vértices pequenos são filhos de $2n$, exceto, possivelmente dois, que tornaram-se isolados. Se este não é o caso, então o algoritmo localiza o valor de f sabendo que $f < 2n + 1$. Basicamente, esta tarefa consiste em verificar as condições (i), (ii) e (iii) do Teorema 6, o que pode ser feito de maneira direta.

Algoritmo 5: Encontra $f(F)$

Entrada: floresta F (uma árvore representativa com duas arestas perdidas)
Saída: elemento fixo $f \leq 2n + 1$

1. **se** F contem um vértice grande x com um irmão z **então**
   retorne $f := max\{x, z\}$
2. **para** cada vértice grande x de F satisfazendo $N_F(x) \neq \emptyset$
   para cada pequeno $y \in N_F(x)$
   $Y' \leftarrow \{x - n, x - n + 1, ..., n\}$
   se ($N_F^*(x) = Y'$ ou $N_F^*(x) \subset Y'$) e ($Y' \setminus N_F^+(x)$ é o conjunto de vértices
   de uma árvore de F) **então**
   retorne $f := x$
3. Encontra o percurso em pré-ordem das 3 árvores de F, e f, o único vértice grande e que está mais à direita do percurso em pré-ordem de alguma árvore de F
   retorne f

Os passos 1 e 3 podem ser facilmente verificados em tempo linear. No passo 2, observe que há, no máximo, dois vértices grandes x de F que satisfazem a condição de ter dois filhos pequenos. Consequentemente, os testes no passo 2 são aplicados a no máximo dois candidatos x, por isso, o algoritmo todo é executado em tempo $O(n)$.

3.4.3 Determinação dos filhos da raiz

Depois de identificar o vértice fixo da marca d'água, temos informações suficientes para determinar as arestas de árvore que foram removidas. Observe que a tarefa é trivial quando $f = 2n + 1$, já que, neste caso, pelo Teorema 1, só pode haver um grafo de permutação redutível canônico G referente a n. Tal grafo possui precisamente uma árvore representativa do Tipo-1, que é única para cada $n > 2$ (Propriedade 3 dos grafos de permutação redutíveis canônicos na Seção 3.2). Pela definição, o percurso em pré-ordem sem raiz de uma árvore representativa do Tipo-1 quando $f = 2n + 1$, é $n + 1, n + 2, ..., 2n, 1, 2, ..., n, 2n + 1$.

Portanto, queremos determinar os filhos de $2n+2$ quando $f < 2n+1$. Seja G uma marca d'água, T sua árvore representativa e F a floresta obtida de T pela remoção de duas arestas. Como de costume, f representa o vértice fixo de T, X representa o conjunto dos vértices grandes diferentes de $2n+2$, e $X_c = X \setminus \{f\}$. Finalmente, denotamos por $A \subseteq X_c$ o subconjunto dos vértices cíclicos grandes ascendentes de T, que, por simplicidade, nos referimos como vértices ascendentes, e denotamos por D o conjunto $D = X_c \setminus A$ de vértices cíclicos grandes descendentes de T, ou simplesmente vértices descendentes. Dada a floresta F e seu vértice fixo f, o Algoritmo 6 computa o conjunto A, que, como verificamos na prova do Teorema 1, corresponde precisamente aos filhos da raiz $2n+2$.

Algoritmo 6: Encontra vértices grandes(F)

Entrada: floresta F (uma árvore representativa com duas arestas removidas)
Saída: os filhos A na raiz $2n+2$ da árvore representativa

1. **se** $F[X_c] \cup 2n+2$ é conexo **então**
   retorne $A := N_F(2n+2)$
2. **se** $F[X_c] \cup 2n+2$ não possui vértices isolados **então**
   retorne $A := N_F(2n+2) \cup 2n+1$
3. **se** $F[X_c] \cup 2n+2$ possui dois vértices isolados x, x' **então**
   retorne $A := N_F(2n+2) \cup \{x, x'\}$
4. **se** $F[X_c] \cup 2n+2$ contêm um único vértice isolado x **então**
   se $|N_F^*(f)| = 2n - f + 1$ **então**
   seja y_r o vértice mais à direita de $N_F^*(f)$
   se $|N_F(2n+2)| < y_r$ **então**
   retorne $A := N_F(2n+2) \cup \{x, 2n+1\}$
   senão
   retorne $A := N_F(2n+2)$
   senão
   retorne $A := N_F(2n+2) \cup \{x\}$

É fácil concluir que o algoritmo pode ser implementado em tempo $O(n)$, portanto provaremos apenas sua corretude.

Teorema 7 *O Algoritmo 6 determina corretamente o conjunto de vértices ascen-*

dentes A de T.

Prova: Vamos analisar as condições que são verificadas pelo algoritmo. Assuma que $F[X_c] \cup \{2n+2\}$ é conexo. Então, $N_T(2n+2) = N_F(2n+2)$, implicando $S = N_F(2n+2)$. Logo, o algoritmo está correto se termina no Passo 1.

Assuma que $F[X_c] \cup \{2n+2\}$ é desconexo, mas não tem vértices isolados. Então, ou $N_F(2n+2) = N_F(2n+2)$ ou a aresta $(2n+2, 2n+1)$ foi uma das que podem ter sido removidas de T. Em qualquer dessas situações, podemos escrever $A = N_F(2n+2) \cup 2n+1$, implicando que o algoritmo também está correto se termina no Passo 2.

Assuma que $F[X_c] \cup \{2n+2\}$ contêm dois vértices isolados distintos x, x'. É possível somente que $x, x' \in N_T(2n+2)$. Então, a construção de A como a união de x, x' e $N_F(2n+2)$ garante a corretude, sempre que o algoritmo termina no Passo 3.

Na última situação, assuma $F[X_c] \cup \{2n+2\}$ contendo um único vértice isolado x. Consideramos as seguintes alternativas. Se $|N_F^+(f)| = 2n - f + 1$, implica que $N_T^*(f) = N_F^*(f)$, porque o conjunto de descendentes de f em T compreende exatamente $y_{f_0}, y_{f_0+1}, ..., y_n$. O número de descendentes de f é definido por $n - f_0 + 1$, que, pela Propriedade 3 dos grafos de permutação redutíveis canônicos é igual a $2n - f + 1$. Agora, pelo Teorema 1, $|N_T(2n+2)| = y_r$, onde y_r é o vértice mais à direita de $N_F^*(f)$. Nesta situação, $|N_F(2n+2)| < y_r$ implica que x necessariamente começa com $N_F(2n+2)$. Adicionalmente, a aresta $(2n+2, 2n+1)$ também foi removida de T, uma vez que uma única remoção de aresta é suficiente para tornar o vértice x isolado. Observe, por outro lado, que isolar um vértice grande que não é filho de $2n+2$ requer a remoção de ao menos duas arestas para $n > 2$. Assim, $A = N_F(2n+2) \cup \{x, 2n+1\}$, e o algoritmo está correto. No caso em que $|N_F(2n+2)| = y_r$, sabemos que $N_T(2n+2) = N_F(2n+2)$, por isso $A = N_F(2n+2)$,

garantindo a corretude do algoritmo. Finalmente, quando $|N_F^*(f)| \neq 2n - f + 1$, significa que alguma aresta de uma subárvore enraizada em f foi removida de T. Neste caso, o vértice isolado x é necessariamente um filho de $2n+2$ em T, implicando em $A = N_F(2n + 2) \cup \{x\}$, e o algoritmo está correto. □

3.4.4 Recuperando arestas removidas

Como conhecemos o conjunto de vértices grandes, podemos recuperar toda a árvore T. Basicamente, dados os conjuntos A e X_c, obtemos o conjunto D de vértices descendentes. Então, de acordo com a ordenação de A e D, podemos localizar todos os vértices cíclicos grandes em T, usando o modelo dado no Teorema 1. Em seguida, f é inserido em T, de modo que seu pai x_q é o menor vértice cíclico maior que f. Finalmente, colocamos os vértices pequenos $\{1, 2, ..., f - n - 1\}$ que são todos filhos de x_n. E os vértices pequenos restantes $\{f - n, f - n + 1, ..., n\}$ são descendentes de f e suas posições em T podem ser obtidas como segue: para cada $y \in \{f - n, f - n + 1, ..., n\}$, encontramos sua posição no percurso em pré-ordem P de T pela determinação do vértice grande x cuja posição na sequência bitônica de vértices cíclicos grandes é exatamente y. Então, y deve ser o x-ésimo vértice no percurso em pré-ordem sem raiz de T. Finalmente, a posição de f em P é claramente igual a f.

Os detalhes são dados no Algoritmo 1, que computa o percurso em pré-ordem P de $T - \{2n + 2\}$.

Novamente, é fácil ver que o Algoritmo 1 computa corretamente o percurso em pré-ordem de T em tempo $O(n)$. Esse procedimento assegura a completa recuperação de T e, portanto, torna possível a recuperação completa da marca d'água G.

3.5 Decodificação em tempo polinomial (k arestas removidas)

O reconhecimento em tempo linear da classe dos grafos de permutação redutíveis canônicos, apresentado na forma do Teorema 1, permite a construção de um algoritmo polinomial para recuperar a marca d'água que sofreu a remoção de k arestas, para um valor arbitrário de k. O algoritmo proposto é formalmente robusto, uma vez que recupera uma marca d'água danificada G' sempre que possível, ou informa que existe mais de uma marca d'água candidata, isto é, que mais de uma marca d'água poderia ter originado a marca d'água danificada (pela remoção de k de suas arestas) O certificado para o último caso, são as duas ou mais marcas d'água retornadas pelo algoritmo que se tornaram isomorfas a G' dada a remoção de exatamente k de suas arestas, provando que nem sempre é possível recuperar a marca d'água.

Seja G um grafo marca d'água e G' o grafo obtido a partir de G quando um certo subconjunto de k arestas são removidas. O algoritmo mais intuitivo — ou ingênuo — para a recuperação de G consiste em simplesmente adicionarmos a G', um por vez, cada um dos possíveis conjuntos de k não-arestas de G', verificando, para cada escolha, se uma marca d'água válida foi assim obtida. Observe que reconhecer se um grafo pertence à classe de grafos associados a marcas d'água de Chroni e Nikolopoulos (isto é, à classe dos grafos de permutação redutíveis canônicos) é tarefa que pode ser realizada em tempo linear, uma vez que o algoritmo é diretamente obtido pela caracterização de tais grafos. Como $|V(G')| = 2n + 3$ e $|E(G')| = 4n + 3 - k$, a quantidade de subconjuntos de k não-arestas de G' é igual a $\binom{\frac{(2n+3)(2n+2)}{2-(4n+3-k)}}{k} = O(n^{2k})$. Portanto, a complexidade total do algoritmo é $O(n^{2k}) \cdot O(n) = O(n^{2k+1})$.

O algoritmo descrito acima é um tanto grosseiro ao considerar que qualquer das não-arestas de uma marca d'água danificada G' poderia ser uma aresta da marca d'água original G. Na verdade, devido à estrutura bem definida — e bastante

restrita — de G, relativamente poucas dentre essas não-arestas de G' poderiam realmente pertencer a G. Mais precisamente, sabemos que quase todos os vértices de G possuem grau de saída exatamente igual a 2, a menos dos vértices de rótulo $2n+2$ e 0. Dessa forma, cada vértice $v \in V(G')$ contribui com $0 \leq 2 - |N^+_{G'}(v)| \leq 2$ elementos para o multiconjunto M^* contendo todos os candidatos a origens de arestas removidas de G. É fácil ver que $|M^*| = 2 \cdot |V(G')| - |E(G')| = 2 \cdot (2n+3) - (4n+3-k) = k+3$, de forma que as k origens podem ser escolhidas de $\binom{|M^*|}{k} = O(k^3)$ maneiras. Uma vez escolhido um subconjunto $M \subseteq M^*$ de k origens, passamos aos vértices de destino. Cada origem tem $O(n)$ destinos possíveis, num total de $O(n^k)$ possibilidades. Considerando a execução do algoritmo linear de reconhecimento para cada uma dessas possíveis escolhas, chegamos à complexidade $O(k^3) \cdot O(n^k) \cdot O(n) = O(k^3 \cdot n^{k+1})$ para todo o algoritmo. Se exatamente uma marca d'água foi reconhecida nesse processo, a recuperação foi bem-sucedida. Caso contrário, terá ficado provada a impossibilidade da recuperação pela exibição de um conjunto $\mathcal{L}$ contendo todas as marcas d'água que poderiam ter originado G'.

Algoritmo 7: Reconstrução da marca d'água

Entrada: grafo $G'(V, E')$
Saída: lista $\mathcal{L}$ de marcas d'água candidatas

1. $\mathcal{L} \leftarrow \emptyset$
2. determine o número de arestas removidas $k \leftarrow 2|V| + 1 - |E|$
3. multiconjunto
$\mathcal{S} \leftarrow \{v \mid v \in V, |N^+_{G'}(v)| = 1\} \cup \{v, v \mid v \in V, |N^+_{G'}(v)| = 0\}$
4. seja $\mathcal{E}' \leftarrow \mathcal{S} \times V$
5. **para cada** conjunto $R = \{e_1, \ldots, e_k\} \subset \mathcal{E}$
5.1. **se** $G = (V, E' \cup R)$ é uma marca d'água **então** adicione G a $\mathcal{L}$
6. **retorne** $\mathcal{L}$ // se $|\mathcal{L}| = 1$, a marca d'água original foi recuperada

O segundo algoritmo que propomos para a recuperação de marcas d'água com k arestas removidas, parte do princípio de que os rótulos de todos os vértices da marca d'água são conhecidos — ou podem ser determinados — *a priori*. Note que, por construção, conhecer os rótulos dos vértices equivale a conhecer todas as

arestas do caminho hamiltoniano de G. Para efeito de análise, portanto, suporemos que as k arestas removidas foram arestas de árvore de G, isto é, arestas que não são da forma $(j, j-1)$ na marca d'água original G.

Sabemos que o vértice de rótulo $2n+2$ é a raiz da árvore representativa T formada por todas as arestas de árvore de G. Uma vez que, por hipótese, k arestas foram removidas de T, o subgrafo F formado pelos vértices de G' (com exceção do vértice de rótulo 0) e por todas as arestas de árvore de G' apresentará exatamente $k+1$ componentes conexos. É também sabido, pela propriedade *max-heap* das árvores representativas (pela qual se u é filho de v em T então $u < v$), que a raiz de cada sub-árvore T' de T é precisamente o vértice de maior rótulo em T'. Ficam, dessa forma, determinadas as origens das arestas removidas de G': os vértices v de maior rótulo em cada componente conexo de F, excetuando-se o componente que contém $2n+2$. Definidas as origens, basta escolher, para cada origem v, uma aresta (v, w), com $w > v$, e acrescentá-la a G', verificando, no final, se o grafo assim obtido constitui uma marca d'água válida.

Algoritmo 8: Reconstrução da marca d'água usando os rótulos

Entrada: grafo $G'(V, E')$
Saída: lista $\mathcal{L}$ de marcas d'água candidatas

1. $\mathcal{L} \leftarrow \emptyset$
2. $E'' \leftarrow E' \setminus \{(2n+2, 2n+1), (2n+1, 2n), \ldots, (1, 0)\}$ // E'' recebe as arestas de árvore
3. $F \leftarrow (V, E'')$
4. $M \leftarrow \emptyset$ // origens das arestas removidas
5. **para cada** componente conexo $C \not\supset \{2n+2\}$ de F
5.1. adicione a M o vértice v de maior rótulo em C
6. **para cada** $R = \{e_1, \ldots, e_k\}$, com $e_i = (v_i, w) \notin E'$, $v_i \in M$, $v_i < w \in V$ $(i = 1, \ldots, k)$
6.1. **se** $G(V, E' \cup R)$ é uma marca d'água **então** adicione G a $\mathcal{L}$
7. **retorne** $\mathcal{L}$ // se $|\mathcal{L}| = 1$, a marca d'água original foi recuperada

Para a análise de complexidade, observe que cada conjunto R possui k ele-

mentos e corresponde a uma atribuição de destinos às origens das arestas removidas. Como cada aresta tem $O(n)$ destinos possíveis, o número de tais atribuições é $O(n^k)$, cada uma das quais correspondendo a um grafo que será testado, em tempo $O(n)$, quanto a ser ou não uma marca d'água válida, perfazendo a complexidade de $O(n^{k+1})$ para todo o algoritmo.

3.6 Resultados computacionais

O objetivo desta seção é o de enriquecer o entendimento do esquema de marca d'água proposto por Chroni e Nikolopoulos [33], observando o comportamento dos algoritmos de decodificação propostos em [33] e o nosso algoritmo apresentado na Seção 3.3, e determinando a probabilidade de que um ataque a k arestas seja bem sucedido. Primeiramente, foram implementados os algoritmos de codificação e de decodificação exatamente como descritos por Chroni e Nikolopoulos [33], assim como o nosso algoritmo de decodificação. Foram, a seguir, geradas as marcas d'água referentes ao conjunto W formado por:

- todas as chaves nos intervalos $[2^4, 2^5 - 1]$ e $[2^9, 2^{10} - 1]$,
- 10 mil chaves escolhidas aleatória e uniformemente no intervalo $[2^{19}, 2^{20} - 1]$,
- 10 mil chaves escolhidas aleatória e uniformemente no intervalo $[2^{29}, 2^{30} - 1]$, e
- 10 mil chaves escolhidas aleatória e uniformemente no intervalo $[2^{99}, 2^{100} - 1]$.

Passemos agora aos testes realizados.[2]

[2]O código-fonte está disponível em `https://www.dropbox.com/s/5tx8h3e720l6bo6/watermarking.py`. Todos os resultados foram obtidos em uma máquina iMac 2.9 GHz Intel Core i5 com 8 GB de RAM.

3.6.1 Comparativo dos algoritmos de decodificação

O primeiro teste considerou a decodificação de marcas d'água íntegras. Com efeito, todas as marcas d'água de W foram utilizadas como entrada para os algoritmos de decodificação estudados. Os dois algoritmos exibiram comportamento nitidamente linear, rodando em tempo $O(n)$ para marcas d'água de tamanho $\theta(n)$, tal como previsto analiticamente. O nosso algoritmo, embora dotado da propriedade de se recuperar de remoções de até 2 arestas, executou marginalmente mais rápido (em torno de 20%) do que o de Chroni e Nikolopoulos [33], que não possui a mesma propriedade. Os resultados podem ser vistos nas duas primeiras colunas da Tabela 3.1.

n (bits)	Chroni et al. [33] (sem remoções)	Nosso algoritmo (sem remoções)	Nosso algoritmo (-1 aresta)	Nosso algoritmo (-2 arestas)
5	82.2 (4.4) μs	56.5 (3.2) μs	63.9 (6.7) μs	78.0 (16.4) μs
10	132.3 (9.3) μs	95.7 (5.8) μs	104.2 (9.4) μs	122.8 (24.8) μs
20	240.9 (11.8) μs	177.5 (9.7) μs	190.7 (17.4) μs	219.9 (44.9) μs
30	357.7 (14.4) μs	268.9 (13.2) μs	281.3 (18.2) μs	328.1 (66.0) μs
100	1406.7 (45.7) μs	1135.4 (39.5) μs	1151.2 (89.8) μs	1248.5 (260.4) μs

Tabela 3.1: Tempos médios (e desvios-padrão) de decodificação.

O segundo teste considerou a decodificação, pelo nosso algoritmo, de marcas d'água das quais $1 \leq k \leq 2$ arestas foram removidas. A partir de um número suficientemente grande de marcas d'água tomadas aleatoriamente de cada um dos subconjuntos de W descritos anteriormente, totalizando um mínimo de 20 mil instâncias por valor de n (ou exaurindo todos os casos possíveis, nos casos $n = 5$ e $n = 10$), foram removidos, um por vez, todos os subconjuntos de $k \in \{1, 2\}$ arestas de cada uma daquelas marcas d'água. Procedemos, a seguir, às decodificações de cada uma das instâncias assim obtidas, e o algoritmo foi capaz de recuperar a marca d'água original em todos os casos testados. A terceira e a quarta coluna

da Tabela 3.1 apresentam os tempos de decodificação médios que foram obtidos, permitindo concluir que foi irrisório o aumento no tempo de execução decorrente de toda a etapa adicional — para a detecção de ataques e reconstituição das marcas d'água atacadas — executada antes da decodificação propriamente dita.

3.6.2 A resiliência da marca d'água de Chroni e Nikolopoulos

O nosso algoritmo de decodificação permite a recuperação de marcas d'água (para todas as chaves com $n > 2$ bits) que sofreram a remoção de até duas arestas. Sabemos que nem sempre é possível recuperar uma marca d'água que sofreu a remoção de três ou mais arestas, uma vez que existem diversos pares de marcas d'água que se distinguem por não mais do que três arestas. Não obstante, para melhor compreendermos a resiliência do esquema considerado a ataques de distorção, importa conhecer a probabilidade de que seja bem-sucedido — no sentido de impedir a recuperação da marca d'água — um ataque no qual $k \geq 3$ arestas são removidas.

Seja $s(n, k)$ o número de marcas d'água G que codificam chaves de tamanho n e para as quais existe alguma marca d'água G' de mesmo tamanho tal que $E_T(G') \setminus E_T(G) = E_T(G) \setminus E_T(G') \leq k$, onde o subscrito T indica que estamos considerando os conjuntos de arestas de árvore apenas (marcas d'água G, G' que satisfazem essa desigualdade são ditas *k-sinônimas*). Seja $r(n, k) = \frac{s(n,k)}{t(n)}$ a razão entre $s(n, k)$ e o total $t(n) = 2^{n-1}$ de marcas d'água distintas para chaves de tamanho n. Finalmente, seja $p(n, k)$ a probabilidade de que a remoção de até k arestas de árvore de uma marca d'água referente a chave de n bits a torne irrecuperável (por deixá-la isomorfa a alguma outra marca com até k arestas de árvore removidas). Considere, para o cálculo de $p(n, k)$, o seguinte experimento:

Experimento A: Escolha, aleatória e uniformemente, uma marca d'água G_j do con-

junto $M_n = \{G_1, G_2, \ldots, G_{t(n)}\}$ de todas as marcas d'água associadas a chaves de tamanho n. Escolha, a seguir, também de forma aleatória e uniforme, um subconjunto K contendo k arestas do conjunto de $2n + 1$ arestas de árvore de G_j.

Queremos obter a probabilidade $p(n, k)$ de que o grafo $G_j \setminus K$ seja isomorfo de $G_{j'} \setminus K'$, para algum grafo $G_{j'} \neq G_j$ pertencente a M_n e algum subconjunto K' de k arestas de árvore de $G_{j'}$. Apesar de ser natural a formulação do experimento acima, os cálculos e a obtenção dos dados experimentais serão facilitados se considerarmos um experimento análogo para a escolha de G_j e K. Seja $\mathcal{K}_i$ a coleção de todos os subconjuntos de k arestas de árvore de G_i, para i de 1 a $t(n)$, e seja $\mathcal{K}$ a união de todos os $\mathcal{K}_i$.

Experimento B: Escolha, aleatória e uniformemente, um conjunto $K \in \mathcal{K}$.

Observe que cada elemento de $\mathcal{K}$ é um subconjunto de k arestas *de um determinado grafo* $G_i \in M_n$. Em outras palavras, uma vez que os conjuntos de vértices dos grafos considerados são mutuamente disjuntos, os $\mathcal{K}_i$ são todos também disjuntos dois a dois. Dessa forma, a escolha de K determina unicamente a marca d'água $G_j \in M_n$ que contém K, e cada marca d'água G_j tem probabilidade idêntica de ser dessa forma escolhida. Em outras palavras, os experimentos A e B são equivalentes.

A probabilidade $p(n, k)$ desejada é, portanto, simplesmente a probabilidade de que o subconjunto K escolhido pelo experimento B seja um subconjunto de k arestas que, quando removido da marca d'água G_j que o contém, torna G_j isomorfa a alguma outra marca d'água de mesmo tamanho e também com k arestas a menos. Definimos $q(n, k)$ como a quantidade de tais subconjuntos. Como K é escolhido uniformemente de $\mathcal{K}$, a probabilidade desejada fica determinada pela razão entre $q(n, k)$ e a quantidade de elementos de $\mathcal{K}$.

O Algoritmo 9, apresentado a seguir, obtém o numerador $q(n,k)$ da razão definida acima. Na linha 2.2, o número binomial representa a quantidade de maneiras de se escolher um conjunto de k arestas de forma a tornar a marca d'água G_j isomorfa à marca d'água $G_{j'}$, que possui exatamente $d \leq k$ arestas distintas das arestas de G_j (fixamos as d arestas, escolhemos livremente as demais); o fator 2 se deve ao fato de serem duas marcas d'água sinônimas, e, portanto, duas contribuições distintas ao somatório $q(n,k)$.

Algoritmo 9: Determinação de $q(n,k)$

1. $q \leftarrow 0$
2. **para todo** par de marcas d'água $G_j, G_{j'} \in M_n$, com $j < j'$
2.1. $d \leftarrow |E_t(G_j) \setminus E_t(G_{j'})|$ // $E_t(G)$ – conj. de arestas de árvore de G
2.2. **se** $d \leq k$ **então** $q \leftarrow q + 2 \cdot \binom{2n+1-d}{k-d}$
3. **retorne** q

Para o denominador da razão em que estamos interessados, podemos escrever $|\mathcal{K}| = t(n) \cdot \binom{2n+1}{k} = 2^{n-1} \cdot \binom{2n+1}{k}$. A probabilidade desejada fica, portanto, determinada por $p(n,k) = \frac{q(n,k)}{|\mathcal{K}|}$, onde numerador e denominador já são conhecidos.

Para o terceiro e último teste, portanto, foram geradas as marcas d'água correspondentes a todas as chaves de n bits e determinados os valores $r(n,k)$ e $p(n,k)$ para todos os pares $(n,k) \in [3,15] \times [2,4]$. Observe que, apesar de terem sido os valores $r(n,k)$ e $p(n,k)$ obtidos considerando-se apenas a remoção de arestas de árvore, esses valores são claramente limites inferiores e superiores, respectivamente, para o caso geral em que k arestas — dentre arestas de árvore e arestas de caminho — foram removidas.

A Tabela 3.2 apresenta os resultados encontrados. Embora a razão $r(n,k)$ aumente com n (o que é natural, dado o crescimento exponencial da quantidade de marcas d'água para chaves de tamanho n, em contraste com o crescimento apenas quadrático do número de possíveis arestas de árvore), a probabilidade de que um

n (bits)	$r(n,3)$	$r(n,4)$	$r(n,5)$	$p(n,3)$	$p(n,4)$	$p(n,5)$
3	50.00%	100.00%	100.00%	5.71429%	8.57143%	33.33333%
4	25.00%	100.00%	100.00%	2.38095%	2.57937%	5.09607%
5	50.00%	93.75%	100.00%	1.21212%	1.93182%	2.91899%
6	68.75%	90.62%	100.00%	0.69930%	1.23876%	1.92696%
7	81.25%	92.19%	98.44%	0.43956%	0.76190%	1.21360%
8	89.06%	94.53%	97.66%	0.29412%	0.47731%	0.75934%
9	93.75%	96.48%	98.05%	0.20640%	0.30999%	0.48427%
10	96.48%	97.85%	98.63%	0.15038%	0.20897%	0.31842%
11	98.05%	98.73%	99.12%	0.11293%	0.14571%	0.21637%
12	98.93%	99.27%	99.46%	0.08696%	0.10463%	0.15169%
13	99.41%	99.58%	99.68%	0.06838%	0.07707%	0.10939%
14	99.68%	99.77%	99.82%	0.05473%	0.05803%	0.08086%
15	99.83%	99.87%	99.90%	0.04449%	0.04453%	0.06108%

Tabela 3.2: Frequência relativa $r(n,k)$ de marcas d'água referentes a chaves de tamanho n e k-sinônimas de alguma outra marca d'água, e probabilidade $p(n,k)$ de uma marca d'água se tornar irrecuperável após a remoção de k arestas.

ataque a k arestas seja bem sucedido decresce com n, o que é extremamente favorável à confiabilidade do esquema considerado.

3.7 Conclusão

A caracterização apresentada na Seção 3.2 nos permite afirmar que os grafos gerados pelo codec proposto por Chroni e Nikolopoulos [32] são apropriados para codificação de uma marca d'água, pois possuem as características discutidas na Seção 2.4. Em contrapartida, o codec produz marcas d'água de tamanho $2n + 3$, onde n é o tamanho da representação binária do identificador, que, conforme demonstramos, são capazes de recuperar ataques de distorção que causam a remoção de, no máximo, duas arestas. Em outras palavras, o codec possui baixa resiliência. Além disso, o codec não apresenta diversidade, pois cada identificador origina apenas

um grafo marca d'água.

4 MARCA D'ÁGUA RANDOMIZADA

Claramente existem diversas formas de codificar um identificador em um grafo apropriado para ser embarcado num programa, a fim de fornecer proteção à propriedade intelectual. Entretanto, não é fácil estabelecer uma relação eficiente entre resiliência, tamanho da marca d'água, furtividade e diversidade.

Como vimos no Capítulo 3, o codec proposto por Chroni e Nikolopoulos produz marcas d'água com $2n + 3$ vértices, onde n é o tamanho da representação binária do identificador. Ou seja, são sempre inseridos $n + 3$ bits de redundância na marca d'água que a tornam capaz de recuperar ataques de distorção que causam a remoção de, no máximo, 2 arestas. Além disso, existe uma bijeção entre identificador e marca d'água produzida, de modo que o codec não apresentada diversidade.

Neste capítulo, apresentaremos um codec que produz marcas d'água com $n + 1$ vértices e possui correspondência de um-para-um entre bits do binário correspondente ao identificador (ou chave) e arestas da marca d'água. Além disso, utiliza randomização para obter diversidade. Isto é, a estrutura da marca d'água é produzida por meio de escolhas aleatórias realizadas durante a execução do algoritmo de codificação. Para prover resiliência ao esquema, propomos o uso de códigos de detecção e correção de erros, amplamente discutidos na literatura. Por simplicidade, vamos considerar que a chave a ser codificada corresponde a um inteiro, mas é importante observar que qualquer sequência de caracteres possui uma representação binária correspondente e, portanto, pode ser codificada pelo codec proposto.

O Capítulo 4 está organizado da seguinte forma: na Seção 4.1, descrevemos os algoritmos propostos para codificação e decodificação da marca d'água, bem como

alguns detalhes do codec. Na Seção 4.2, apresentamos uma possível implementação linear para o algoritmo de codificação proposto na Seção 4.1. E, na Seção 4.3, descrevemos como podemos utilizar os códigos conhecidos de detecção e correção de erros para prover resiliência à marca d'água.

4.1 A marca d'água randomizada

Definimos um codec que possui as seguintes propriedades:

- O algoritmo de codificação gera as arestas da marca d'água de maneira randomizada, permitindo que uma mesma chave gere marcas d'água distintas para diferentes execuções do algoritmo;
- Existe uma correspondênca entre arestas da marca d'água e bits da chave codificada, permitindo que ataques de distorção não sejam propagados, sejam detectados após a decodificação e corrigidos (para um número definido) por algum algoritmo padrão de correção de erros previamente definido;
- A marca d'água produzida pelo codec possui $n + 1$ vértices e, no máximo, $2n + 2$ arestas, onde n corresponde ao tamanho do binário do identificador;
- Os algoritmos de codificação e decodificação podem ser implementados em tempo linear.

Os passos do algoritmo de codificação são descritos no Algoritmo 10. Observe que o algoritmo recebe como entrada uma chave ω e produz como saída um grafo marca d'água G correspondente com $n+1$ vértices e, no máximo, $2n-1$ arestas (divididas em arestas de retorno e arestas de caminho), onde n corresponde ao número

de bits na representação binária B de ω. Além disso, se $C : v_1, v_2, \ldots, v_d, v_{d+1} = v_1$ é um ciclo com d vértices do grafo G, dizemos que os vértices $v_2, \ldots, v_{d-1}$ são vértices internos de C.

Algoritmo 10: Codificação da marca d'água randomizada

Entrada: uma chave inteira ω
Saída: uma marca d'água randomizada codificada G

1. Seja B a representação binária de ω, e seja $n = |B|$. Atribua índices aos bits de B, da esquerda para a direita, iniciando em 1.

2. A marca d'água $G(V, E)$ é inicialmente isomorfa a um caminho direcionado P_{n+1} com vértices $V = \{1, \ldots, n+1\}$, isto é, o conjunto E inicialmente contém arestas de caminho de v para $w = v+1$, denotada $[v \rightarrow w]$, para $1 \leq v \leq n$.

3. Para cada vértice $v \in V \setminus \{1, n+1\}$, adicione em E uma aresta de retorno de v para w, denotada $[w \leftarrow v]$, onde w é escolhido uniformemente de maneira aleatória entre os vértices de V que satisfazem:
   - w não é um vértice interno de um ciclo de G, e
   - $v - w$ é um inteiro positivo ímpar se v é índice de um bit 1 em B, ou par se v é índice de um bit 0 em B.

   Se w não existir, então v possui grau de saída 1, ou seja, v não possui aresta de retorno.

A Figura 4.1 mostra duas marcas d'água distintas geradas pelo Algoritmo 10 ao receber como entrada a chave $\omega = 395$, no qual o binário obtido no passo 1 do algoritmo é $B = 110001011$, com $n = 9$ bits. As duas marcas d'água possuem o mesmo número de vértices, ou seja, $n + 1$ e ambas têm um único caminho hamiltoniano criado no passo 2 do algoritmo. O primeiro vértice do caminho hamiltoniano, com rótulo 1, sempre corresponde a um 1 em B, e sempre possui grau de saída igual a 1. Cada vértice de 2 a n torna-se origem de zero ou uma aresta de retorno. As arestas de retorno com origem em cada $v \in \{2, \ldots, n\}$ (caso exista) terá uma correspondência de um-para-um com os bits indexados de 2 até n em B: um bit

1 com índice $v \geq 2$ em B origina uma aresta de retorno $[w \leftarrow v]$ com um salto ímpar através do caminho hamiltoniano, enquanto que um bit 0 origina uma aresta de retorno com salto par através do caminho hamiltoniano, se existir (ou seja, uma aresta de retorno não é criada quando não existe um vértice $w < v$, tal que $v - w$ é par e w não é um vértice interno de um ciclo). Agora, considere o vértice 2 que corresponde a um bit 1 em B e, portanto, torna-se origem da aresta de retorno $[1 \leftarrow 2]$, a única escolha possível no momento. O vértice 3, por sua vez, corresponde a um 0 e torna-se origem da aresta de retorno $[1 \leftarrow 3]$, a única escolha possível. Agora o vértice 4, que corresponde a um 0, deve possuir uma aresta de retorno para um $w < 4$ tal que $4 - w$ seja par, isto é, $w = 2$, mas 2 é um vértice interno do ciclo $1, 2, 3, 1$, e esta aresta não pode ser criada. O vértice 5 corresponde a um bit 0 e agora duas arestas de retorno são possíveis, a aresta $[1 \leftarrow 5]$ e a aresta $[3 \leftarrow 5]$. A Figura 4.1 (a) mostra a escolha da aresta $[1 \leftarrow 5]$ e a Figura 4.1 (b) corresponde a marca d'água obtida após a escolha da aresta $[3 \leftarrow 5]$. O algoritmo gera as arestas de retorno de maneira semelhante para os vértices $6, \ldots, 9$, completando a marca d'água.

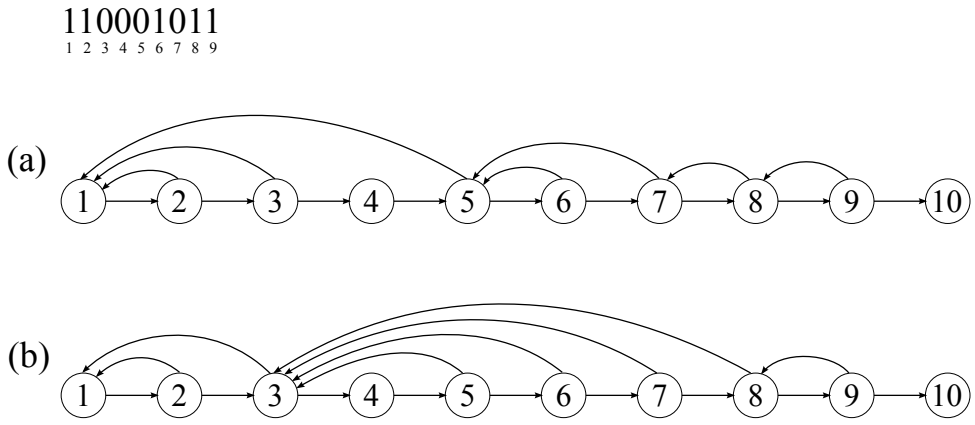


Figura 4.1: Marcas d'água distintas codificando a chave $\omega = 395$

Note que é possível que todos os vértices do grafo possuam grau de saída igual a 2, exceto os vértices 1 e $n + 1$ que possuem grau de saída igual a 1 e 0,

respectivamente. Além disso, o caminho hamiltoniano oferece uma ordenação dos vértices. Assim, de acordo com o que vimos na Seção 2.4, podemos concluir que o grafo produzido pelo algoritmo de codificação pode ser considerado apropriado.

O Algoritmo 11 descreve os passos da decodificação da marca d'água e consiste de dois passos. O primeiro passo efetua a rotulação dos vértices da marca d'água para que os bits do binário codificado possam ser determinado. Tal rotulação sempre pode ser realizada, já que os blocos básicos do caminho hamiltoniano estão sempre consecutivos no CFG, correspondendo aos vértices $1, 2, \ldots, n+1$. No segundo passo do Algoritmo 11 é definido o primeiro bit do binário B como 1 (o que é sempre verdade, uma vez que zeros à esquerda de um número são ignorados), e começamos a identificar os demais bits através das arestas de retorno. Ou seja, a partir do vértice 2, uma aresta de retorno $[w \leftarrow v]$, tal que $v - w$ é ímpar, indica que o bit com índice v no binário B é 1, e, analogamente, uma aresta de retorno $[w \leftarrow v]$, tal que $v - w$ é par, indica que o bit com índice v no binário B é 0. Vértices $2 \leq v \leq n$ que não são origem de arestas de retorno correspondem a bits 0 em B. Observe que se houver um ataque de distorção causando a remoção de alguma aresta da marca d'água, apenas o bit de índice igual ao vértice origem da aresta removida será decodificado (correta ou incorretamente) como 0, não afetando a decodificação dos demais bits da chave codificada.

Algoritmo 11: Decodificação da marca d'água randomizada

Entrada: uma marca d'água randomizada G com $n+1$ vértices
Saída: a chave ω codificada em G

1. Rotule os vértices de G em ordem crescente ao longo do caminho hamiltoniano de G.
2. Seja B um *array* de bits iniciando num bit 1 seguido por $n-1$ bits 0.
3. Para cada vértice $v \in \{2, n\}$, se há um vértice $w < v$ tal que $[w \leftarrow v] \in E(G)$ e $v - w$ é ímpar, então $B[v] \leftarrow 1$; caso contrário, $B[v] \leftarrow$ '0'.
4. Retorne $\omega = \sum_{i=1}^{n} B[i] \cdot 2^{n-i}$.

O Algoritmo 11 efetua a rotulação dos vértices do grafo percorrendo-os apenas uma vez, e identifica os bits codificados nas arestas percorrendo as listas de adjacência dos vértices $2, \ldots, n$ somente uma vez. Logo, o Algoritmo 11 claramente pode ser implementado em tempo linear.

4.2 Implemetação da codificação em tempo linear

Nesta Seção, veremos uma possível implementação em tempo linear para o algoritmo de codificação da marca d'água. Note que os dois primeiros passos do Algoritmo 10 são diretos. Para implementar o passo 3, entretanto, deve-se levar em consideração todos os candidatos a destino, isto é, vértices $w < v$ que não sejam internos a qualquer ciclo e que possam ser destino de uma aresta de retorno com origem no vértice v que está sendo processado. Se v é par e corresponde a um bit 0, então o destino w da aresta de retorno $[w \leftarrow v]$ é selecionado entre os candidatos a destino chamados pares, ou se v é ímpar e corresponde a um bit 1, então o destino w da aresta de retorno $[w \leftarrow v]$ é selecionado entre os candidatos a destino chamados ímpares. Em todos os casos, o algoritmo escolhe de maneira aleatória e uniforme entre os possíveis vértices de destino nomeados de acordo com a paridade, o que pode ser feito selecionando um inteiro entre 1 e o número de candidatos.

Vamos utilizar duas pilhas S_0 e S_1, cada uma implementada em um *array* tal que qualquer item pode ser buscado pelo seu índice em tempo constante. Implementando tais pilhas com *arrays*, o método *pop_ all(i)* leva tempo constante para remover todos os itens que possuem índices que são maiores que um dado índice i.

A proposta de implementação do passo 3 consiste de um *loop* que percorre os vértices $2, ..., n$ na ordem determinada pelas arestas de retorno, uma por uma. As seguintes afirmações são verdadeiras: os elementos em S_0 que possuem rótulos

pares são candidatos a destino em ordem ascendente (de cima para baixo ao longo da pilha) naquele momento; analogamente, os elementos em S_1 que possuem rótulos impares são candidatos a destino em ordem ascendente naquele momento. Todos os vértices $v = 1, ..., n$ serão adicionados em suas respectivas pilhas (vértices de rótulos pares em S_0, vértices de rótulos ímpares em S_1) exatamente uma vez durante a execução do algoritmo, isto é, até o final da iteração na qual v é visitado (logo após a definição do destino da aresta de retorno com origem no vértice v).

Além disso, será necessário utilizar um *array* auxiliar com tamanho n, que chamaremos de A, inicialmente vazio, e cujas posições possuem índices de 1 a n. Para cada posição v do *array*, quando v for par, será atribuído o tamanho que a pilha S_1 tinha quando v foi adicionado na pilha S_0. Analogamente, em cada posição v do *array*, se v é ímpar, será atribuído o tamanho que a pilha S_0 tinha no momento em que v foi adicionado em S_1.

Agora podemos descrever a implementação em tempo linear para o passo 3 do algoritmo de codificação (Algoritmo 10). Os detalhes são apresentados como pseudocódigo no Algoritmo 12.

Depois da inicialização das estruturas de dados (linha 1), o vértice $v = 1$ é o primeiro a ser considerado. No entanto, uma vez que não há arestas de retorno com origem no vértice 1 para ser adicionado, o algoritmo apenas empilha v em S_1 (porque v é ímpar) e escreve 0 (o tamanho atual da pilha S_0) na posição 1 de A (linha 2). Agora, para cada vértice $v \in \{2, ..., n\}$, o algoritmo primeiro decide qual pilha — S_0 ou S_1 — contem os candidatos possíveis a destino da aresta de retorno com origem em v que será (randomizadamente) selecionado (linhas 4-7). Tal pilha será referenciada como S, sendo S_0 se houver um salto de distância par (quando $B[v]$ é um bit 0) e v é par, ou um salto de distância ímpar é necessário (quando $B[v]$ é 1) e v é ímpar; caso contrário, a pilha apropriada será $S = S_1$. O elemento em

Algoritmo 12: Determinação das arestas de retorno em $O(n)$

Entrada: o n-bit da representação binária de B da chave codificada e o conjunto E contendo apenas arestas de caminho da marca d'água
Saída: conjunto E atualizado contendo todas as arestas da marca d'água

```
1.   S_0 := pilha vazia; S_1 := pilha vazia; A := vetor com n zeros
2.   S_1.push(1); A[1] := 0
3.   para v = 1, ..., n faça
4.       se (v é par e B[v] = 0) ou (v é ímpar e B[v] = 1) então
5.           S := S_0; S' := S_1
6.       senão
7.           S := S_1; S' := S_0
8.       se S.tamanho > 0 então
9.           j := inteiro escolhido de maneira aleatória e uniforme entre
                  [1, S.tamanho]
10.          w := S[j]
11.          E := E ∪ {[w ← v]}
12.          S.pop_all(j)
13.          S'.pop_all(A[w])
14.      se v é par então
15.          S_0.push(v); A[v] := S_1.tamanho
16.      senão
17.          S_1.push(v); A[v] := S_0.tamanho
18.  retorne E
```

$\{S_0, S_1\} \setminus S$ será referenciado como S'. Se S é vazio, então não há vértices possíveis que podem ser destinos de arestas de retorno partindo de v; neste caso, nenhuma aresta de retorno será adicionada ao conjunto E. Por outro lado, um inteiro j é escolhido uniformemente de maneira randomizada entre 1 e $|S|$, para determinar o destino $w = S[j]$ da aresta de retorno com origem em v. Esta aresta de retorno é adicionada em E (linha 11).

Agora, como a adição da aresta de retorno $[w \leftarrow v]$ implica na criação do ciclo $C : w, w+1, ..., w+(v-w) = v, w$ todos os vértices internos de C não poderão ser usados como possíveis candidatos a vértices de destino no futuro. Em outras

palavras, os vértices internos de C deixam de ser candidatos de destino, e podem ser removidos de suas pilhas correspondentes, isto é, todos os vértices $w' > w$ devem ser removidos de S e S'. Uma vez que o índice de w em S, chamado j, é conhecido, é fácil remover todos os vértices w' de S (linha 12), que correspondem aos vértices que possuem índice maior que j em S. Por outro lado, como w nunca será um elemento de S', não há vértice com índice w em S', e podemos realizar uma busca (binária) em S' para determinar o índice do último elemento de S' que é menor que w — que adicionaria um fator extra $O(\log n)$ na complexidade do algoritmo. No entanto, como w não é um vértice interno de um ciclo (caso contrário, não teria sido escolhido como destino de uma aresta de retorno), nenhum vértice $w' < w$ foi selecionado como destino depois que w foi processado, o que significa que todos os vértices que pertencem a S' na hora em que w estava sendo processado ainda pertencem a S' e permanecem lá. Como consequência deste fato, apenas estes vértices permanecem em S', uma vez que vértices adicionados em S' depois de w ser processado são necessariamente maiores que w (e menores que v), portanto estes vértices se tornam vértices internos do ciclo C e devem ser removidos de S'. Em outras palavras, S' deve possuir precisamente os r primeiros elementos, onde r é o tamanho que S' possuía quando w foi processado e adicionado em S. É nesse momento que fazemos uso do *array* auxiliar A, pois $r = A[w]$ é precisamente o valor que foi armazenado na w-ésima posição quando w foi adicionado em S (linha 15 ou 17, depende da paridade de w).

Uma vez que existe um número constantes de operações por vértice, executadas claramente em tempo $O(1)$, todo o Algoritmo 12 possui complexidade $O(n)$.

4.3 Resiliência a ataques de distorção

A resiliência do esquema de marca d'água proposto é obtida com o uso de códigos de detecção/correção de erros. Existe uma vasta literatura que trata dessas técnicas, principalmente no contexto de trocas de mensagens em canais sujeitos a erros [47, 48, 49, 50, 51]. De modo que, não pretendemos apresentar uma discussão sobre as técnicas existentes. Em vez disso, vamos considerá-las apenas como "caixas-preta", demonstrando como tais técnicas podem prover resiliência ao esquema proposto.

Embora as técnicas de correção de erros possam ser diferentes (e muito) na maneira de tratar um binário possivelmente danificado, um requisito comum é a inserção de um número $f(n,t)$ de bits de redundância, para alguma função f. Sendo assim, a inserção de tais bits de redundância será realizada numa etapa de pré-processamento da representação binária da chave a ser codificada.

Na fase de decodificação do nosso esquema de marca d'água, o efeito de k remoções de arestas é o equivalente a escrever erroneamente k ou menos bits 0 nas posições ocupadas originalmente por bits 1 no binário codificado. Isto ocorre porque quando não há uma aresta de retorno com origem no vértice v, para $2 \leq v \leq n$, o decodificador considera que há um bit 0 no índice v. Se uma aresta de retorno com origem em v existia na marca d'água antes do ataque, então o bit com o índice v no binário original poderia ser 1. Assim, a consequência de cada remoção de aresta é a mudança de um único bit, pois, devido a mecânica do codec proposto, os erros decodificados não são propagados.

Por outro lado, suponha que em vez de ser baseado na paridade da distância percorrida por saltos invertidos ao longo do caminho hamiltoniano, o nosso algoritmo de codificação seleciona o destino w de uma aresta de retorno com origem em v, da

seguinte forma: escolhe de maneira aleatória e uniforme, um destino $w < v$ tal que w não é um vértice interno de um ciclo, e w corresponde a um bit no binário que é o mesmo bit que está na n-ésima posição. Em outras palavras, se v é um 1, a aresta de retorno que deixa v é um 0. Neste codec, uma remoção de aresta, resulta na decodificação errada de um vértice v com transferência de erro em cascata para o vértice v' que é saída da aresta de retorno que chegou em v, e para o vértice v que é saída da aresta de retorno que chega em v', e assim por diante.

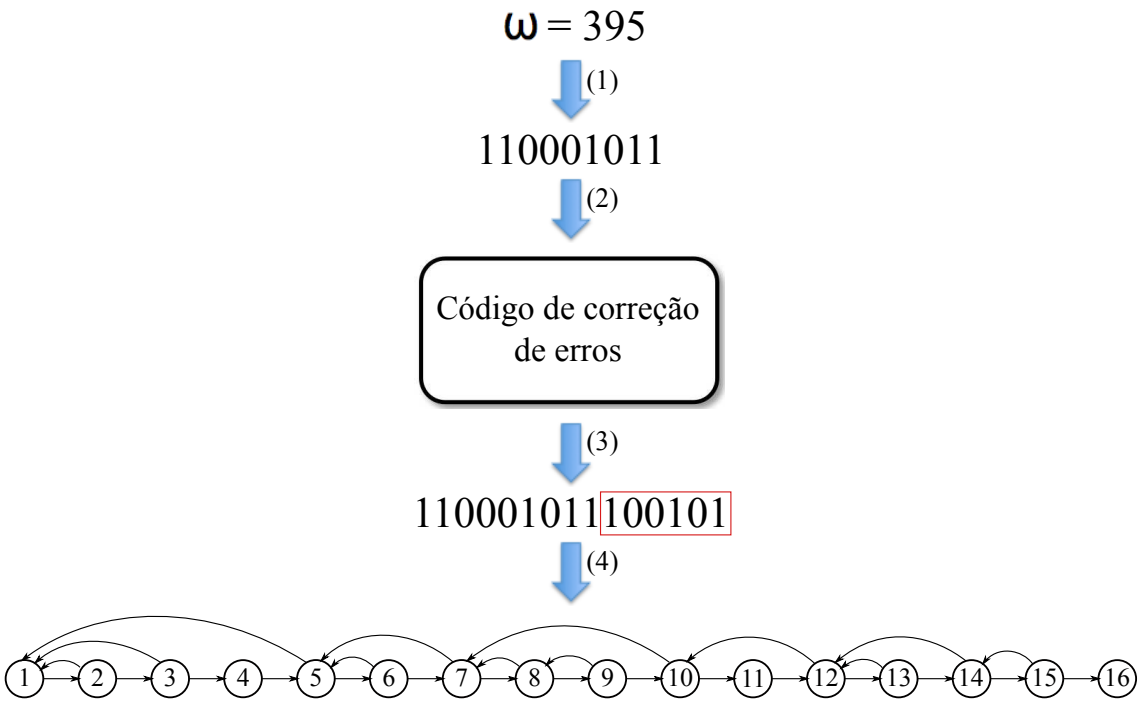


Figura 4.2: Codificação de $\omega = 395$ com correção de 1 bit

Ilustramos na Figura 4.2 a codificação da mesma chave apresentada na Figura 4.1, mas agora usando a técnica de correção de erro conhecida como Reed-Solomon [50] sob um campo de Galois $GF(2^3)$, que, neste caso, fornece a capacidade de se recuperar um bit (isto é, uma remoção de aresta). No passo (1), o binário é obtido; nos passos (2) e (3), o binário é trasmitido para uma etapa de pré-processamento de correção de erros onde os bits de redundância são adequada-

mente inseridos; no passo (4), o binário final é codificado no grafo marca d'água usando o algoritmo de codificação proposto (Seção 4.1, Algoritmo 10). Note que no passo de pré-processamento pode ser definido um número arbitrário $t > 0$ de *flips* de arestas que possam ser detectados e corrigidos, resultando no aumento da dimensão do binário.

A decodificação é feita de forma semelhante, conforme ilustrado na Figura 4.3: no passo (1), o grafo marca d'água que sofreu a remoção da aresta $[5 \leftarrow 6]$ é decodificado, obtendo-se o binário representante (Seção 4.1, Algoritmo 11); nos passos (2) e (3), o binário decodificado é passado para o passo de correção de erro, de onde outro binário (com os bits flipados corrigidos) é produzido; e, finalmente, no passo (4), a chave original é recuperada.

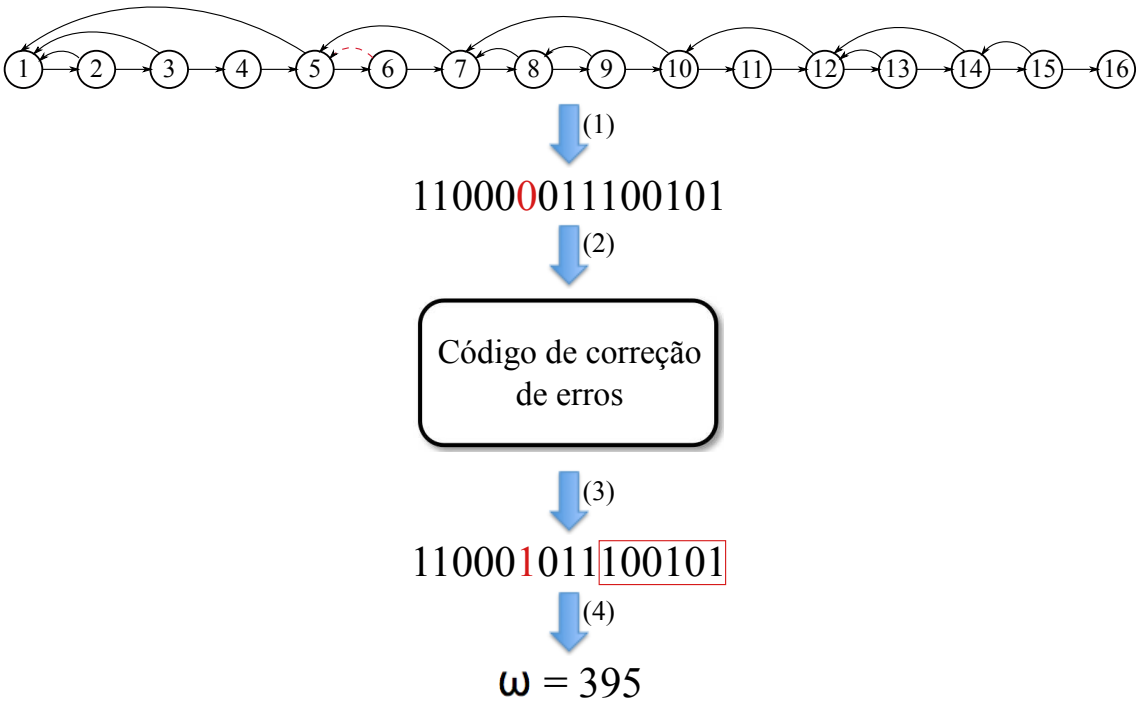


Figura 4.3: Decodificação da marca d'água com sucesso

Assim, se o número k de arestas removidas for menor ou igual ao limite fixo $t > 0$ levado em consideração durante o pré-processamento do binário original, o

método de correção de erros utilizado pelo codec deve identificar e corrigir os bits flipados; caso contrário, o ataque conseguirá danificar a marca d'água permanentemente. De fato, não importa a técnica de correção de erros escolhida ou o número de t erros suportados, o atacante sempre pode remover um número tão grande $t' > t$ de arestas que não possa ser recuperado. A Figura 4.4 ilustra um caso onde houve a remoção das arestas $[5 \leftarrow 6]$ e $[8 \leftarrow 9]$ da marca d'água codificada na Figura 4.2. Como na etapa de pré-processamento foram inseridos bits de redundância suficientes para recuperar a remoção de até uma aresta, a técnica de detecção/correção de erros não foi capaz de recuperar os bits flipados e, com isso, o algoritmo de decodificação produziu um identificador ω incorreto.

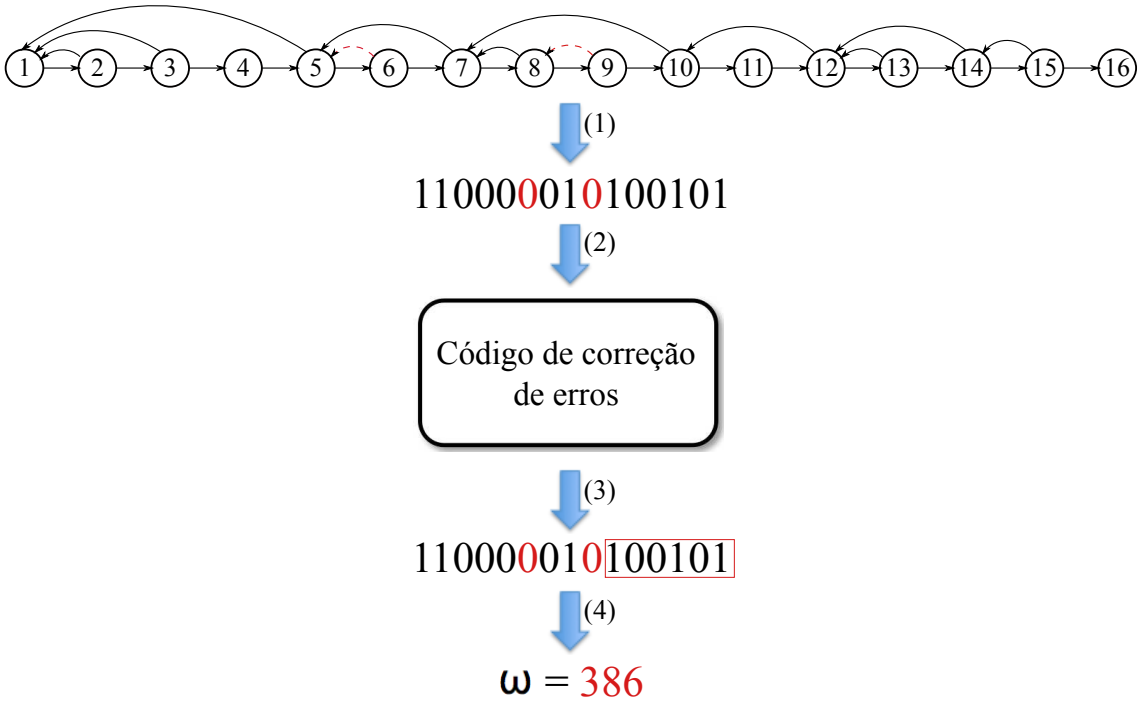


Figura 4.4: Decodificação da marca d'água com erro

4.4 Propriedades da marca d'água randomizada

A concepção de um codec de marca d'água deve levar em consideração diversos cenários de ataque e técnicas utilizadas para embarcar a marca d'água no programa a ser protegido, de modo que seja possível garantir sua eficácia na proteção antipirataria. Assim, conforme veremos a seguir, o codec proposto na Seção 4.1 provê as principais propriedades discutidas na literatura.

4.4.1 Diversidade

Como vimos, no contexto de marca d'água baseada em grafos, chamamos de diversidade a capacidade de gerar grafos marca d'água razoavelmente distintos para codificar a mesma chave [5].

As arestas das marcas d'água geradas pelo codec proposto são criadas de forma aleatória, o que significa que a estrutura do grafo marca d'água é produzida de acordo com escolhas aletórias que ocorrem durante a execução do algoritmo de codificação. Assim, a mesma chave pode gerar grafos distintos. Em resumo, essa característica permite embarcar múltiplas cópias da marca d'água no programa a ser protegido, tornando menos provável que uma marca d'água seja identificada (e consequentemente atacada) por meio de comparações de força bruta — realizadas por ferramentas de *diff* — entre diferentes programas com marca d'água do mesmo autor, dificultando, assim, a realização dos ataques de subtração e distorção.

Para obter sucesso ao realizar um ataque de subtração, o atacante deverá encontrar todas as cópias da marca d'água e removê-las. O fato dos grafos gerados pelo codec serem considerados apropriados para a codificação de marcas d'água, aumenta a dificuldade para localizar todas as marcas d'água embarcadas no programa

e, consequentemente, dificulta a realização de ataques de subtração.

O ataque de distorção também torna-se árduo, pois além de encontrar dificuldade para localizar todas as cópias de marca d'água inseridas no programa, o atacante precisará realizar mudanças no código do programa de modo que todas as cópias de marca d'água sejam danificadas o suficiente para que nenhuma delas possa ser recuperada pela técnica de detecção/correção de erros utilizada.

4.4.2 Resiliência

No codec proposto, existe uma bijeção entre arestas da marca d'água e bits da chave e um processo de decodificação imune à propagação de erros, tornando possível o uso de técnicas padrão de detecção/correção de erros [48, 49, 50] de forma bastante direta. Assim, o número de arestas removidas que nosso codec é capaz de suportar diante de um ataque de distorção pode ser personalizado. Em outras palavras, nosso codec pode possuir resiliência a ataques de distorção para um número k de remoções de arestas tão grande quanto seja de interesse do proprietário do programa a ser protegido.

4.4.3 Taxa de dados

A taxa de dados expressa a quantidade de dados que podem ser escondidos no interior de uma marca d'água [25, 5]. De acordo com o processo de codificação, as marcas d'água produzidas pelo nosso codec possui $n+1$ vértices, onde n é o tamanho do binário correspondente à chave. Após a etapa de pré-processamento realizada para efetuar a inclusão dos bits de redundância, o tamanho da marca d'água torna-se $r + n + 1$, onde r é o número de bits de redundância utilizados para recuperação

de t remoções de arestas mal-intencionadas.

A quantidade de bits de redundância depende da técnica de correção de erros adotada e do número de bits que pretende-se recuperar diante de ataques de distorção. Entretanto, em aplicações onde a resiliência pode ser dispensada, a marca d'água embarcada no programa a ser protegido terá o tamanho original, ou seja, $n+1$ vértices e, no máximo, $2n+2$ arestas. Em outros casos, onde o programa será executado em ambientes hostis e o uso de técnicas de detecção/correção de erros sejam indispensáveis para prover resiliência á marca d'água, pode-se utilizar técnicas que embarcam partes da marca d'água em diferentes pontos do programa [28]. Isto é, a marca d'água é fragmentada e cada fragmento é inserido em diferentes pontos do programa. Esta alternativa ajuda a disfarçar a marca d'água no programa, mesmo que sejam utilizados muitos bits de redundância, além de ajudar a evitar diferentes tipos de ataque [5].

4.4.4 Furtividade

A furtividade refere-se ao fato de que a marca d'água não pode ser facilmente reconhecida quando embarcada em programas típicos. Este conceito é bastante importante no contexto de marca d'água baseada em grafos, pois se uma marca d'água tiver estrutura diferente de um CFG, o atacante pode facilmente encontrá-la e removê-la.

De acordo com o que vimos na Seção 2.4, um grafo marca d'água deve possuir grau de saída máximo baixo e uma ordenação fornecida pelas arestas de saída. Isso significa que se considerarmos um esquema de marca d'água estático, deve ser possível traduzir o grafo marca d'água em código-fonte, onde a ordenação fornecida pelas arestas de saída correspondem a sequência das instruções no código, e o

grau de saída baixo indica as ligações entre as instruções, também conhecidas como instruções de salto (que normalmente existem poucas em cada bloco). Em outras palavras, o grafo marca d'água deve corresponder a um CFG.

O Algoritmo de codificação do nosso codec (Algoritmo 10) cria um caminho hamiltoniano P_{n+1} do vértice 1 ao vértice $n+1$ e, em seguida, cria arestas de retorno deixando cada vértice $v \in V \setminus \{1, n+1\}$. Assim, os vértices das marcas d'água produzidas por nosso codec possuem grau de saída 0, 1 ou 2 e, portanto, podem coincidir com grafos de fluxos de controle gerados a partir de programas reais.

4.5 Conclusão

O codec proposto é capaz de codificar uma mesma chave em diferentes grafos marca d'água, provendo alta diversidade, uma característica destacada como sendo de grande importância pela comunidade [5]. Os grafos produzidos pelo algoritmo de codificação possuem uma ordenação dos vértices definida pelo caminho hamiltoniano e grau de saída máximo dois, podendo ser, portanto, considerados apropriados, já que se assemelham ao grafo de fluxo de um programa real. Além disso, devido a correspondência de um-para-um entre arestas de retorno e bits da representação binária da chave, os ataques de distorção não se propagam, tornando possível o uso de técnicas conhecidas de detecção/correção de erros para prover resiliência parametrizável.

Note que as características observadas no codec proposto independem dos algoritmos *embarcador* e *extrator* utilizados para embarcar e extrair a marca d'água do programa. Entretanto, a escolha correta dos mesmos pode tornar o esquema de marca d'água ainda mais eficaz na proteção antipirataria.

5 CONCLUSÕES

Com o crescimento do setor de tecnologia, a demanda por software tem se tornado cada vez maior, reforçando o interesse em técnicas de proteção de software que ajudem a garantir os direitos autorais e a defesa antipirataria, como técnicas de marca d'água, por exemplo. Nesta tese, estudamos alguns conceitos de marca d'água de software, em especial marcas d'água de software baseadas em grafos e aprofundamos nossos estudos no codec proposto por Chroni e Nikoloupolos, caracterizando a classe dos grafos gerados por tal codec. Com base nesta caracterização, realizamos um estudo da resiliência das marcas d'água produzidas pelo codec e fornecemos algoritmos para recuperação das marcas d'água diante de ataques de distorção. Além disso, apresentamos alguns resultados que demonstram que este codec é pouco resiliente diante deste tipo de ataque.

Para solucionar algumas deficiências identificadas no codec proposto por Chroni e Nikolopoulos, apresentamos um codec que possui todas as características discutidas na Seção 2.4:

- Nosso codec pode ser implementado em tempo linear;
- Os grafos produzidos por nosso codec são apropriados para serem usados como marca d'água;
- Os grafos produzidos possuem $n + 1$ vértices e, no máximo, $2n + 2$ arestas. Este tamanho pode ser considerado pequeno, devido as condições que devem ser satisfeitas para o grafo ser considerado apropriado;
- A utilização de randomização para criação das arestas de retorno da marca

d'água permite que um identificador origine mais de um grafo marca d'água;

- Incrementando o tamanho da marca d'água com bits de redundância (definidos por algum algoritmo de detecção/correção de erros) é possível obter resiliência parametrizável.

Dando continuidade aos nossos estudos, estamos trabalhando na caracterização formal dos grafos de fluxo de programas descritos por meio do que é conhecido como *programação estruturada*. A programação estruturada pode ser vista como um método para desenvolvimento de algoritmos e programas. Basicamente, consiste num conjunto de padrões de qualidade que torna o programa mais detalhado, legível, confiável e de fácil manutenção, amplamente utlizado na descrição de programas na atualidade. Uma versão preliminar da caracterização dos grafos de programas estruturados foi descrita no trabalho intitulado "The Graphs of Structured Programming", apresentado no 13th Cologne-Twente Workshop on Graphs and Combinatorial Optimization, 2015.

Com a caracterização dos grafos de programas estruturados será possível propor um codec de marca d'água demonstradamente furtivo. Ou seja, conhecendo bem a estrutura do grafo de fluxo de um programa estruturado, proporemos um codec capaz de gerar grafos que possuem as mesmas características que o grafo de fluxo destes programas. Em outras palavras, os grafos gerados pelo novo codec pertencerão a classe dos grafos de programas estruturados. Assim, ao considerar um esquema de marca d'água estático, o código correspondente ao grafo gerado por este codec será estruturado e, portanto, difícil de ser localizado por um atacante durante a realização de uma análise estática[1]. Além disso, se a marca d'água for embarcada com cuidado e o código-fonte correspondente implementar alguma funcionalidade do programa, tornará a marca d'água mais difícil de ser encontrada durante uma

[1]Análise estática: método de depuração de programas feito através da análise do código sem executar o programa. Pode utilizar ferramentas, como o grafo de fluxo do programa, por exemplo.

análise dinâmica.

O novo codec manterá as características já apresentadas pelo codec randomizado, isto é, as arestas serão criadas utilizando randomização para prover diversidade, a marca d'água gerada possuirá tamanho pequeno, a resiliência do codec será obtida pelo uso de técnicas de detecção/correção de erros e os algoritmos de codificação e decodificação poderão ser implementados em tempo linear.

Além disso, para completar o esquema de marca d'água estruturado, vamos propor os algoritmos *embarcador* e *extrator* de modo que as características obtidas pelo codec sejam preservadas e realçadas.

REFERÊNCIAS

[1] THE SOFTWARE ALLIANCE. **Comunicados antipirataria**. Disponível em: <http://www.bsa.org/anti-piracy/ap-communications/?sc_lang=pt-BR>. Acesso em: 20 ago. 2015.

[2] BSA GLOBAL SOFTWARE SURVEY. **Security threats rank as top reason not to use unlicensed software**. Disponível em: <http://globalstudy.bsa.org/2013/>. Acesso em: 25 ago. 2015.

[3] ZHU, W. F. **Concepts and techniques in software watermarking and obfuscation**. New Zealand: The Department of Computer Sciences, 2007.

[4] CRAIG, P.; BURNETT, M. Cackers. In: CRAIG, P.; HONICK, R.; BURNETT, M. **Software piracy exposed**. [S. l.]: Syngress, 2005. cap. 4, p. 61-94.

[5] COLLBERG, C.; NAGRA, J. **Surreptitious software**: obfuscation, watermarking, and tamperproofing for software protection. Upper Saddle River: Addison-Wesley, 2010.

[6] AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers**: principles, techniques, and tools. Massachusetts: Addison-Wesley, 1986.

[7] DAVIDSON, R. I.; MYHRVOLD, N. **Method and system for generating and auditing a signature for a computer program**. US 5559884 A, 24 sep. 1996.

[8] QU, G.; POTKONJAK, M. Analysis of watermarking techniques for graph coloring problem. In: COMPUTER-AIDED DESIGN, 1998, San Jose. **Proceedings**... New York: IEEE, 1998, p. 190-193.

[9] MYLES, G.; COLLBERG, C. Software watermarking through register allocation: implementation, analysis, and attacks. In: LIM, J.-I.; LEE, D.-H. (Eds.). **Information security and cryptology - ICISC 2003**. Berlin: Springer, 2004. p. 274-293. (Lecture Notes in Computer Science, 2971).

[10] ZHU, W.; THOMBORSON, C.; WANG, F.-Y. A survey of software watermarking. In: KANTOR, P. et al (Ed.). **Intelligence and security informatics**. Berlin: Springer, 2005. p. 454-458. (Lecture Notes in Computer Science, 3495).

[11] LEE, H.; KANEKO, K. New approaches for software watermarking by register allocation. In: SOFTWARE ENGINEERING, ARTIFICIAL INTELLIGENCE, NETWORKING, AND PARALLEL/DISTRIBUTED COMPUTING, 9., 2008, Phuket. **Proceedings**... New York: IEEE, 2008. p. 63-68.

[12] LI, J.; LIU, Q. Design of a software watermarking algorithm based on register allocation. In: E-BUSINESS AND INFORMATION SYSTEM SECURITY, 2., 2010, Wuhan. **Proceedings**... New York: IEEE, 2010. p. 1-4.

[13] STERN, J. P. et al. Robust object watermarking: application to code. In: PFITZMANN, A. (Ed.). **Information hiding**. Berlin: Springer, 2000. p. 368–378. (Lecture Notes in Computer Science, 1768).

[14] COLLBERG, C. S.; SAHOO, T. R. Software watermarking in the frequency domain: implementation, analysis, and attacks. **Journal of Computer Security**, v. 13, n. 5, p. 721-755, 2005.

[15] ZHANG, X.; ZHANG, Z.; ZHANG, C. Spread spectrum-based fragile software watermarking. In: NORTH-EAST ASIA SYMPOSIUM ON NANO, INFORMATION TECHNOLOGY AND RELIABILITY, 15., 2011, Macao. **Proceedings**... New York: IEEE, 2011. p. 150-154.

[16] FENG, T. et al. A spread spectrum watermarking algorithm based on local instruction statistic. In: INTELLIGENT INFORMATION HIDING AND MULTIMEDIA SIGNAL PROCESSING, 9., 2013, Beijing. **Proceedings**... New York: IEEE, 2013. p. 551-554.

[17] MONDEN, A. et al. A practical method for watermarking Java programs. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 24., 2000, Taipei. **Proceedings**... New York: IEEE, 2000. p. 191-197.

[18] FUKUSHIMA, K.; SAKURAI, K. A software fingerprinting scheme for Java using classfiles obfuscation. In: CHAE, K.-J.; YUNG, M. (Ed.). **Information security applications**. Berlin: Springer, 2004. p. 303-316. (Lecture Notes in Computer Science, 2908).

[19] ARBOIT, G. A method for watermarking java programs via opaque predicates. In: INTERNATIONAL CONFERENCE ON ELECTRONIC COMMERCE AND RESEARCH, 5., 2002, Montreal. **Proceedings**... 2002. p. 23-27.

[20] COLLBERG, C.; MYLES, G.; HUNTWORK, A. Sandmark–a tool for software protection research. **IEEE Security and Privacy**, Piscataway, v. 1, n. 4, p. 40-49, 2003.

[21] COUSOT, P.; COUSOT, R. An abstract interpretation-based framework for software watermarking. In: SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 31., 2004, Venice. **Proceedings**... New York: ACM, 2004. p. 173-185.

[22] VENKATESAN, R.; VAZIRANI, V.; SINHA, S. A graph theoretic approach to software watermarking. In: MOSKOWITZ, I. S. (Ed.). **Information hiding**. Berlin: Springer, 2001. p. 157-168. (Lecture Notes in Computer Science, 2137).

[23] VENKATESAN, R.; VAZIRANI, V. **Technique for producing through watermarking highly tamper-resistant executable code and resulting watermarking code so formed**, US 7051208 B2, 23 may 2006.

[24] COLLBERG, C. et al. Graph-based approaches to software watermarking. In: BODLAENDER, H. L. (Ed.). **Graph-theoretic concepts in computer science**. Berlin: Springer, 2003. p. 156-167. (Lecture Notes in Computer Science, 2880).

[25] COLLBERG, C. et al. Dynamic path-based software watermarking. In: PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 2004, Washington. **Proceedings**... New York: ACM, 2004. p. 107-118.

[26] COLLBERG, C. et al. More on graph theoretic software watermarks: implementation, analysis, and attacks. **Information and Software Technology**, v. 51, n. 1, p. 56-67, 2009.

[27] COLLBERG, C. S.; THOMBORSON, C.; TOWNSEND, G. M. Dynamic graph-based software fingerprinting. **ACM Transactions on Programming Languages and Systems**, v. 29, n. 6, oct. 2007.

[28] COLLBERG, C.; THOMBORSON, C. Software watermarking: models and dynamic embeddings. In: ACM SIGPLAN-SIGACT SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 26., 1999, Texas. **Proceedings**… New York: ACM, 1999. p. 311-324.

[29] CHRONI, M.; NIKOLOPOULOS, S. D. **Efficient encoding of watermark numbers as reducible permutation graphs**. Epirus: Department of Computer Science, University of Ioannina, 2011. 20 p. (Technical Report TR-2011-03).

[30] _____. Encoding watermark integers as self-inverting permutations. In: COMPUTER SYSTEMS AND TECHNOLOGIES, 11., 2010, Sofia. **Proceedings**… New York: ACM, 2010. p. 125-130.

[31] _____. Encoding watermark numbers as cographs using self-inverting permutations. In: COMPUTER SYSTEMS AND TECHNOLOGIES, 12., 2011, VienNa. **Proceedings**... New York: ACM, 2011. p. 142-148.

[32] _____. An efficient graph codec system for software watermarking. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE WORKSHOPS, 36., 2012, Izmir. **Proceedings**… New York: IEEE, 2012. p. 595-600.

[33] _____. An embedding graph-based model for software watermarking. In: INTELLIGENT INFORMATION HIDING AND MULTIMEDIA SIGNAL PROCESSING, 8., 2012, Piraeus. **Proceedings**… New York: IEEE, 2012. p. 261-264.

[34] _____. Multiple encoding of a watermark number into reducible permutation graphs using cotrees. In: COMPUTER SYSTEMS AND TECHNOLOGIES, 13., 2012, Ruse. **Proceedings**… New York: ACM, 2012. p. 118-125.

[35] _____. Design and evaluation of a graph codec system for software watermarking. In: DATA TECHNOLOGIES AND APPLICATIONS, 2., 2013, Reykjavik. **Proceedings**… 2013. p. 277-284.

[36] BENTO, L. M. S. et al. Towards a provably resilient scheme for graph-based watermarking. In: BRANDSTäDT, A.; JANSEN, K.; REISCHUK, R. (Ed.). **Graph-Theoretic Concepts in Computer Science**. Berlin: Springer, 2013. p. 50-63. (Lecture Notes in Computer Science, 8163).

[37] _____. **Full characterization of a class of graphs tailored for software watermarking**. Disponível em:

<http://viniciusgusmao.com/manuscripts/watermarking_JCSS_1.pdf>. Acesso em: 14 set. 2015.

[38] _____. Grafos de permutação redutíveis canônicos: caracterização, reconhecimento e aplicação a marcas d'água digitais. In: LATIN-AMERICAN WORKSHOP ON CLIQUES IN GRAPHS, 6., 2014, Pirenópolis. **Proceedings**… 2014. p. 26.

[39] _____. Proteção de software por marcas d'água baseadas em grafos. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 40., 2013, Maceió. **Proceedings**... 2013.

[40] _____. **On the resilience of canonical reducible permutation graphs and their suitability for software watermarking**. Disponível em: <http://viniciusgusmao.com/manuscripts/watermarking_JCSS_2.pdf>. Acesso em: 14 set. 2015.

[41] _____. A randomized graph-based scheme for software watermarking. In: SIMPÓSIO BRASILEIRO DE SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS, 24., 2014, Belo Horizonte. **Anais**... Porto Alegre: Sociedade Brasileira de Computação, 2014. p. 30-41.

[42] HAMILTON, J.; DANICIC, S. A survey of static software watermarking. In: WORLD CONGRESS ON INTERNET SECURITY, 2011, London. **Proceedings**... New York: IEEE, 2011. p. 100-107.

[43] HECHT, M. S.; ULLMAN, J. D. Flow graph reducibility. In: ANNUAL ACM SYMPOSIUM ON THEORY OF COMPUTING, 4., 1972, Denver. **Proceedings**... New York: ACM, 1972. p. 238-250.

[44] _____. Characterizations of reducible flow graphs. **Journal of the ACM**, New York, v. 21, n. 3, p. 367-375, July, 1974.

[45] TARJAN, R. Testing flow graph reducibility. In: SYMPOSIUM ON THEORY OF COMPUTING, 5., 1973, Texas. **Proceedings**... New York: ACM, 1973. p. 96-107.

[46] VERNET, O.; MARKENZON, L. Hamiltonian problems for reducible flowgraphs. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY, 17., 1997, Valparaiso. **Proceedings**... New York: IEEE, 1997. p. 264-267.

[47] HAMMING, R. W. Error detecting and error correcting codes. **Bell System Technical Journal**, v. 29, n. 2, p. 147-160, 1950.

[48] MANN, H. B. (Ed.). **Error correcting codes**: proceedings of a symposium. New York: Wiley, 1968.

[49] PURSER, M. **Introduction to error-correcting codes**. Boston: Artech House, 1995.

[50] REED, I. S.; SOLOMON, G. Polynomial codes over certain finite fields. **Journal of the Society for Industrial and Applied Mathematics**, v. 8, n. 2, p. 300-304, 1960.

[51] WICKER, S. B. **Error control systems for digital communication and storage**. Englewood Cliffs: Prentice-Hall, 1995.