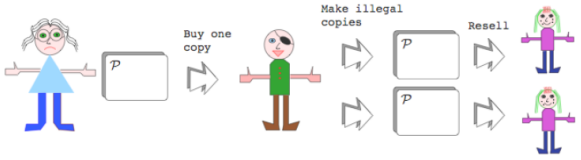
Software Watermarking

Abbiamo detto, che le tecniche di protezione del codice sono essenzialmente tre:

1. **Offuscamento**;
2. **Tampering**;
3. **Watermarking** del codice → esso ha come obiettivo principale quello di contrastare la pirateria, dove per pirateria intendiamo: l'utilizzo di codice e/o di frammenti di codice proprietario in modo illecito.



Vogliamo quindi evitare che Bob, dopo aver comprato una copia dell'applicazione di Alice (ovvero la proprietaria del codice), rivenda a sua volta le copie dell'applicazione a terze parti e quindi vogliamo evitare, che Bob rivenda l'applicazione come propria → l'idea alla base del Watermarking, è quella di inserire e conseguente nascondere all'interno del codice dell'applicazione un'informazione, che identifichi in maniera univoca il reale proprietario del codice dell'applicazione.

> 💡 Il **Watermarking**, quindi, inserisce informazioni riguardo alla proprietà del codice e conseguentemente identifica in maniera univoca il proprietario del codice.
> Si parla, invece, di **Fingerprint** quando si identifica in maniera univoca sia il proprietario sia l'acquirente.

Abbiamo detto, che il Watermarking inserisce e nasconde informazioni all'interno del codice. Tale inserimento deve godere di alcune caratteristiche fondamentali, ovvero:

- **credibilità** → dovrebbe essere facile codificare la paternità del codice, riducendo al minimo la probabilità di coincidenza con altri messaggi;
- **data-rate** → massimizzare la quantità di bit di informazioni che possono essere incorporati all'interno dell'algoritmo di Watermarking;

- **protezione delle parti** → il Watermark dovrebbe essere distribuito uniformemente nel codice, proteggendo in questo modo tutte le sue parti e conseguentemente rendere più complicata l'eliminazione (da parte dell'attaccante) della porzione di codice che codifica l'informazione di proprietà;
- **invisibilità percettiva** (**stealth**) → per l'attaccante dovrebbe essere complicato individuare la porzione di codice che codifica l'informazione di proprietà e di conseguenza dovrebbe presentare le stesse proprietà del codice che lo circonda;
- **resilience** → il Watermark dovrebbe essere resistente a una varietà di attacchi e in questo senso, misuriamo la resistenza in base a:
  - un **attacco sottrattivo** → l'attaccante tenta di rimuovere tutto o parte del Watermark dal programma.

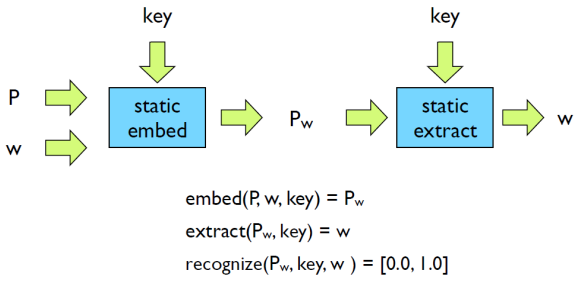
Abbiamo, quindi, il programma originale *P* e lo trasformiamo nel programma *P'*, il quale contiene la firma *W*. L'attaccante sottrattivo vuole, in un qualche modo, trasformare *P'* in un *P"*, in cui la firma *W* non è più presente (ovviamente, come avviene anche con l'offuscamento, tutti i programma *P*, *P'* e *P"* devono mantenere la stessa semantica).

  - un **attacco additivo** → l'attaccante tenta di aggiungere un nuovo Watermark. In realtà, aggiungere un altro Watermark è sempre possibile e di conseguenza bisogna capire, se è possibile proteggersi attraverso altre informazioni;
  - un **attacco distorsivi** → l'attaccante applica trasformazioni al programma firmato, in modo tale da impedire l'estrazione del Watermark. Per dimostrare la resistenza contro gli attacchi distorsivi, solitamente si prova ad **offuscare** il programma marcato, dato che gli attacchi distorsivi vanno ad effettuare delle trasformazioni al programma marcato, mantenendone però la semantica, e cercando di rompere il Watermark. Il modo più semplice, come abbiamo detto, è di utilizzare degli algoritmi di offuscamento, che magari vanno a rompere delle invarianti su cui lavora l'algoritmo di Watermark e di conseguenza la firma non è più estraibile oppure viene estratta la firma sbagliata;

- un **attacco collusivo** → l'attaccante compara due programmi marcati e cerca di capire se vi sono delle parti comuni tra i due programmi, le quali (ovvero le parti comuni) possono corrispondere all'algoritmo di Watermark e una volta individuato, l'attaccante riesce ad eliminarlo.

> 💡 Il Watermark, quindi, **scoraggia** le copie e la ridistribuzione illegale del programma. Inoltre, il Fingerprint può essere utilizzato per rintracciare la fonte della ridistribuzione illegale. Da notare, però, che né il Watermark né il Fingerprint impediscono la copia e la ridistribuzione illegale.

Vediamo, a questo punto, il funzionamento dell'algoritmo di Watermark:



possiamo notare immediatamente, che l'algoritmo di Watermark è costituito da due algoritmi:

- **algoritmo di embedding** → esso prende in ingresso:
  - il programma originale *P*, che vogliamo firmare;
  - la firma *W*;
  - una chiave *key*, che cerca di rendere non sufficiente per l'attaccante conoscere solamente l'algoritmo di embedding.
- **algoritmo di extraction** → una volta che abbiamo il programma marcato, per riuscire ad estrarre la firma utilizziamo un algoritmo di extraction. Di conseguenza, l'agoritmo di extraction prende in ingresso:
  - il programma marcato *Pw*;
  - la chiave *key*, che è conosciuta solamente dal legittimo proprietario del Watermark.

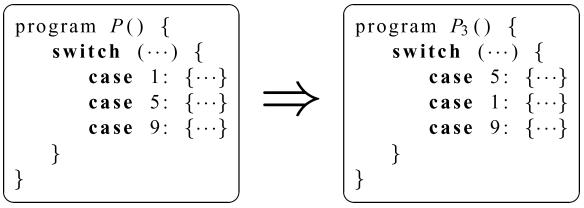
Chiaramente, la credibilità è data dalla probabilità di riuscire ad estrarre la firma *W*, la quale deve essere molto alta quando il programma è *Pw* (ovvero il programma

marcato con la firma *W*) e molto bassa in tutti gli altri casi.

A questo punto sorge spontanea la domanda: “**Come possiamo inserire il Watermark all’interno del codice?**” Chiaramente, se andiamo ad inserire il Watermark direttamente come una parte di testo del codice (attraverso, ad esempio, un commento) non è né stealthy né resistente alle varie tipologie di attacco e di conseguenza non può essere una metodologia da adottare. Inizialmente, allora, il Watermark viene memorizzato nell’eseguibile stesso del programma. Ad esempio, può essere memorizzato direttamente nei **dati** oppure nel **codice** di un file eseguibile e in questo caso, possiamo parlare rispettivamente di

- **Static Data Watermarks**, il quale quindi:
  - viene incluso nella stringa di dati inizializzata;
  - facile da inserire ed estrarre, ma è molto sensibile agli attacchi distorsivi e agli attacchi sottrattivi.
- **Static Code Watermarks**, il quale quindi:
  - la firma è inserita nella sezione di testo (codice) del programma in maniera ridondante;
  - viene utilizzato la maggior parte delle volte nelle immagini e negli audio. Infatti, si parla molto spesso di Media Watermark, in cui il Watermark è comunemente incorporato nei bit ridondanti, che **non** possiamo rilevare a causa dell'imperfezione della nostra percezione umana.

Sorge a questo punto spontanea la domanda: “**Come funziona effettivamente il Watermark statico?**” Il codice del programma ha delle zone di ridondanza (esattamente come ci sono delle zone di ridondanza nelle immagini e nei file audio), dove per zone di ridondanza intendiamo quelle zone in cui il programma calcola lo stesso valore, ma sono scritte in maniera diversa (ovvero sono delle porzioni di codice, che calcolano lo stesso valore, ma sono sintatticamente diverse, in quanto sono scritte in maniera diversa) → tali zone di ridondanza, permettono di scrivere lo stesso programma (ovvero che calcola lo stesso risultato) in modi sintatticamente diversi e di conseguenza, si cercherà di inserire dei bit di informazione nella scelta di una particolare tipologia di zona di ridondanza. I primi algoritmi di Watermark, quindi, funzionavano seguendo questo principio: Ogni volta che in un programma vi sono dei costrutti (o statement del programma), che possono essere in un qualche modo riordinati, nella scelta di una tipologia di ordinamento possono essere inseriti dei bit di informazione (in particolare, possono essere inseriti O(m*log(m)) bit di informazione). Per capire meglio questo concetto, vediamo un esempio:



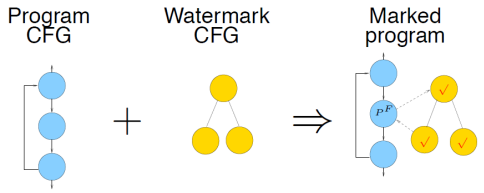
Vediamo che abbiamo uno **Switch** e ad ogni **case** (dello switch) associamo un'informazione e nella scelta dell'ordine dei case andiamo ad inserire dei bit di informazione.

Tendenzialmente il riordino dei costrutti è stealthy, ma **il collo di bottiglia di questo approccio è che è poco resistente agli attacchi distruttivi, in quanto l'attaccante può permutare in modo casuale ogni istruzione dello Switch, in modo da cancellare tutti i mark inseriti**.

→ le prime idee di algoritmi di Watermark, quindi, consistevano nell'utilizzare:

- algoritmi di Media Watermark già esistenti;
- riordinare i costrutti (o statement) del programma.

Un'altra **famiglia di algoritmi di Watermarking lavora sul Control Flow Graph** → l'idea alla base di questa famiglia di algoritmi era quella di codificare l'informazione all'interno di un grafo, ovvero era quella di trovare un modo per codificare un numero in un grafo e successivamente dobbiamo far corrispondere il grafo (in cui abbiamo codificato il numero) nel Control Flow Graph del programma, ovvero dobbiamo realizzare un programma che utilizzi il grafo (in cui abbiamo codificato il numero) come Control Flow Graph. Capiamo immediatamente, che i**l problema di questa soluzione è di riuscire ad agganciare il Control Flow Graph che incorpora il Watermark, con il Control Flow Graph del programma che si vuole marcare**. Il nostro obiettivo, riassumendo, è quello di riuscire ad attaccare al Control Flow Graph del programma originale, il Control Flow Graph del programma che codifica il Watermark, in modo tale da ottenere il Control Flow Graph del programma marcato. Possiamo vedere meglio questo concetto, attraverso la seguente figura:



Generalmente, i Control Flow Graph che incorporano il Watermark devono godere delle seguenti caratteristiche:

- deve essere **riducibile**;

- devono essere relativamente **piccoli**;
- devono essere **resistenti** all'**edge-flip** → l'edge flip è una trasformazione del codice, che ne preserva la semantica, e consiste nell'andare a negare tutte le guardie presenti nel codice e di conseguenza inverte il ramo vero con quello falso.

Capiamo, allora, che per rendere resistente il Control Flow Graph del programma marcato, dobbiamo fare in modo che distinguere tra il Control Flow Graph del programma originale e il Control Flow Graph del programma che codifica il Watermark sia molto difficile (ovviamente, maggiormente i due Control Flow Graph sono attaccanti, maggiore sarà la resistenza del Control Flow Graph del programma marcato) → dobbiamo, però, considerare che l'algoritmo di estrazione (dato che ci ricordiamo, che gli algoritmi di Watermark sono composti da un algoritmi di Embedded e un algoritmo di estrazione) deve essere in grado di distinguere i nodi dei due Control Flow Graphs e di conseguenza, devono essere inseriti dei **marks**, che indicano quali sono i nodi:

- del Control Flow Graph del programma originale;
- del Control Flow Graph del programma che codifica il Watermark.

> 💡 Chiaramente l'inserimento di tali marks deve essere fatto con attenzione, sia perchè senza di essi l'algoritmo di estrazione non è in grado di distinguere i nodi dei due Control Flow Graphs, sia perchè è possibile che i marks vengano identificati dall'attaccante.

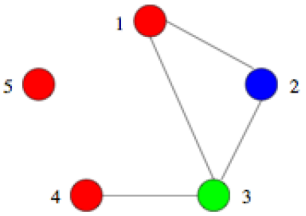
Un altro algoritmo di Watermarking molto conosciuto, è l'**algoritmo QP** → tale algoritmo consiste nell'andare a lavorare sulla locazione dei registri. L'algoritmo QP si basa sul **grafo di interferenza** → il grafo di interferenza viene utilizzato, in fase di compilazione, per allocare le variabili nei registri e modella la relazione tra le variabili di un programma ed in particolar modo, dato un programma, il grafo di inferenza ha:

- un **nodo** per ogni variabile;
- un **arco** tra due variabili, se vi è almeno un punto nel programma, in cui le due variabili sono entrambe "**vive**". Una variabile è "viva" in un certo momento, se contiene un valore che può essere necessario in futuro, o equivalentemente se il suo valore può essere letto prima della prossima scrittura della variabile → chiaramente, se le due variabili sono entrambe vive nello stesso punto del

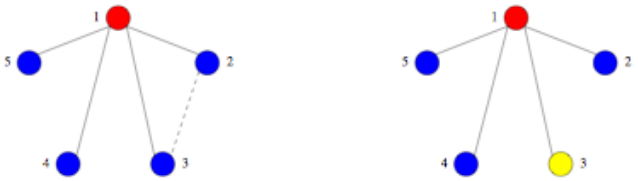
codice, non potranno utilizzare lo stesso registro, mentre se non sono vive entrambe nello stesso punto, possono utilizzare lo stesso registro.

L'algoritmo QP, quindi, funziona nel seguente modo: Dato un programma, vado a costruire il grafo delle interferenza, il quale ha un nodo per ogni variabile del programma. Una volta creati i vari nodi del grafo, vado a fare l'analisi di liveness, in modo tale da istanziare i vari archi del grafo. A questo punto, devo risolvere il **graph coloring problem**, ovvero: Una volta istanziati gli archi, devo andare a colorare con il minor numero di colori possibili (dato che tipicamente si vuole utilizzare il minor numero di registri possibili), in modo tale che tra due nodi tra cui esiste un arco, non abbiano mai lo stesso colore (il nostro obiettivo, quindi, è andare a colorare i nodi del grafo, in modo tale che due nodi collegati da un arco, non abbiano lo stesso colore). Per capire meglio questo concetto, vediamo un esempio:

```
V1 := 2 * 2
V2 := 2 * 3
V3 := 2 * v2
V4 := v1 + v2
V5 := 3 * v3
        mult R1, 2, 2
        mult R2, 2, 3
        mult R3, 2, R2
        add R1, R1, R2
        mult R1, 3, R3
```



Sorge a questo punto spontanea la domanda: "**Come possiamo utilizzare il graph coloring per aggiungere il Watermark?**" L'idea è di fare finta che vi sia un arco aggiuntivo, che in realtà non esiste → aggiungere un nuovo arco ha come conseguenza diretta avere una diversa locazione dei registri. Vediamo un esempio, per capire meglio questo concetto:



Vediamo nel grafo a sinistra, che i vertici 2-3-4-5 possono essere allocati nello stesso registro, in quanto non vi è alcun arco tra di loro. Ipotizziamo, allora, di aggiungere un arco immaginario tra i vertici 2 e 3. L'aggiunta di questo arco comporta il fatto, che i vertici 2 e 3 devono essere di colore diverso, in quanto non possono essere allocati nello stesso registro → questo esempio, ci ha fatto capire, che il graph coloring

inserisce dei vincoli, che in realtà non ci sono, per andare a modificare l'allocazione dei registri. Successivamente, in fase di estrazione, ci troveremo una colorazione che non corrisponde alla colorazione corretta (in quanto abbiamo utilizzato un colore in più), andremmo ad estrarre uno zero oppure un uno.

> 💡 Possiamo, quindi, andare ad utilizzare il graph coloring per aggiungere dei vincoli e nell'aggiunta di tali vincoli, andiamo ad inserire dell'informazione. L'algoritmo QP, quindi, va ad aggiungere informazione andando a cambiare l'allocazione dei registri e successivamente l'algoritmo di estrazione è in grado di estrarre l'informazione.

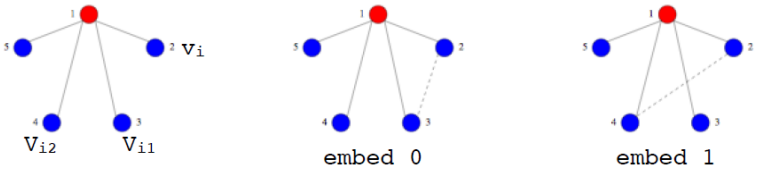
Ci viene immediatamente un'altra domanda:"**Quando possiamo inserire questi archi immaginari?**" Per aggiungere gli archi immaginari, dobbiamo innanzitutto avere tre nodi (quindi una tripla di nodi) che hanno:

- lo stesso colore e di conseguenza non hanno archi tra di loro.

"**Tra quali nodi, però, inserisco l'arco immaginario?**" Per capire dove inserire l'arco, devo andare a numerare tutti i vertici del grafo, in modo tale da ottenere un ordinamento dei vertici. A questo punto, vado ad individuare le triple di vertici con lo stesso colore e supponiamo che:

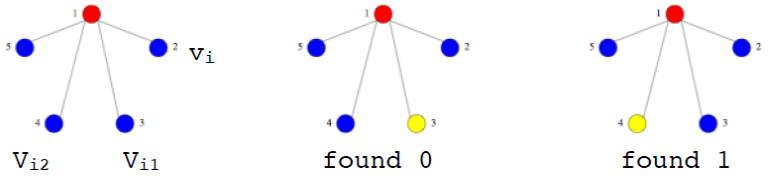
i2 > i1 > i ovvero che il valore del vertice 2 sia maggiore del valore del vertice 1, il quale a sua volta è maggiore del valore del vertice i.

Otteniamo, allora, le seguenti informazioni:



se aggiungiamo un arco tra *V1* e *Vi*, allora diciamo che abbiamo inserito uno ***zero*** di informazione. Invece, se aggiungiamo un arco tra *V2* e *Vi*, allora diciamo che abbiamo inserito un ***uno*** di informazione.

Naturalmente in fase di estrazione, ci ritroveremo che in un caso sarà *V1* ad avere un colore diverso da *Vi*, mentre nell'altro caso sarà *V2* ad avere un colore diverso da *Vi*:



Capiamo, allora, che l'algoritmo QP ha:

- una velocità di trasmissione dei dati molto bassa (low data-rate) → in quanto, il bit rate è dato dal numero di triplette di vertici, che possono essere allocati nello stesso registro;
- **facile da attaccare**, in quanto è sufficiente una permutazione di registro;
- è **abbastanza stealthy** , in quanto è difficile invertire l'embedding.

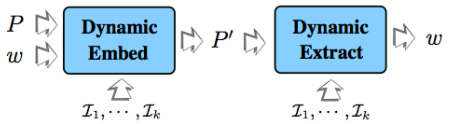
Tutti gli algoritmi di Watermark visti fino a questo momento, vengono detti algoritmi di Watermark statici, in quanto le fasi di embedding ed estrazione si basano su un'analisi statica del programma → **il problema principale di tutti gli algoritmi di Watermark statici è l'offuscamento** (in quanto ricordiamoci, che l'offuscamento funziona molto bene contro l'analisi statica), dato che l'offuscamento distrugge il flusso di controllo, andando ad inserire nuovi rami e nuovi blocchi.

Per risolvere questo problema, possiamo andare a definire un **Watermarking dinamico** → la famiglia di algoritmi di Watermark dinamici vanno ad inserire il Watermark, non in aspetti statici del codice (come avviene negli algoritmi di Watermark statici), bensì va ad inserire il Watermark nella semantica del programma, ovverosia gli algoritmi di Watermark dinamici fanno sì, che il Watermark sia qualcosa che faccio calcolare al programma. Questo ha come conseguenza diretta, il fatto che il Watermark diventa più resistente, dato che l'attaccante distorsivo non deve distorcere gli aspetti statici o la sintassi del programma, bensì deve distorcere la semantica del programma e chiaramente distorcere la semantica e al tempo stesso mantenerla, è molto complicato.

Nel Watermark dinamico abbiamo sempre una fase di embedding e una fase di estrazione, ma rispetto a prima, la fase di embedding riceve in input:

- il programma originale;
- la firma *W*;
- una serie di **input**.

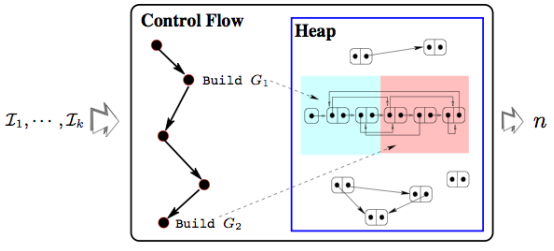
La fase di estrazione, allo stesso modo, prenderà in input il programma marcato ed eseguendolo su quella particolare serie di input, riuscirà ad estrarre il Watermark.



Possiamo osservare, che negli algoritmi di Watermark statici avevamo la chiave *key*, mentre negli algoritmi di Watermark dinamici, la chiave key è stata sostituita dall'input *I* → questo succede, perchè il programma marcato eseguirà il Watermark solamente quando il programma marcato verrà eseguito su quel particolare input. Capiamo, allora, che vi sarà una parte di codice che verrà sempre calcolata (ovvero verrà calcolata qualsiasi sia l'input), mentre quando il programma verrà stimolato con un determinato input, esso (ovvero il programma) andrà a calcolare il Watermark.

Tipicamente, il Watermark viene inserito in una proprietà del codice (quindi in una proprietà che si può analizzare sulle tracce del codice) difficile da analizzare per l'attaccante. A questo punto, vediamo tre esempi di Watermarking dinamico:

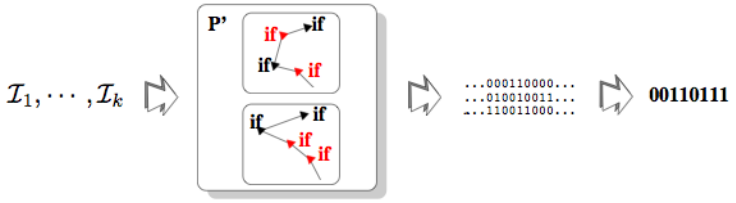
1. **Dynamic Data-structure watermark** → consiste nell'andare ad incorporare il Watermark nelle strutture dati dinamiche utilizzate dal programma. Tale esempio è stato sviluppato da Collberg e l'idea alla base consiste nell'avere una sequenza di input, la quale (ovvero la sequenza di input) va ad attivare una particolare traccia del Control Flow del programma e mano a mano che vado ad eseguire la traccia, sull'Heap (ovvero la parte dinamica della memoria) vado a costruire la struttura dati dinamica (nel senso, che mano a mano che eseguiamo la traccia, andiamo ad esempio a costruire un pezzo della struttura dati, poi ne distruggiamo un pezzo, poi potremmo sovrascrivere un pezzo e così via). Ad un certo punto dell'esecuzione della traccia (da notare, che tale punto non deve essere necessariamente la fine dell'esecuzione della traccia e di conseguenza, il punto può essere un segreto da proteggere dall'attaccante), se andiamo a decodificare la forma che ha struttura dati dinamica sull'Heap otterremo il Watermark.



Questo esempio di Watermarking dinamico è:

- stealthy → in quanto il codice assomiglia al codice standard che si trova in molte applicazioni reali ricche di puntatori;
- resistente → in quanto l'attaccante potrebbe offuscare il codice per distruggere il Watermark, inserendo aggiornamenti distruttivi che non modificano la semantica del programma e quindi teoricamente sarebbe debole contro aggiornamenti distruttivi, ma sono veramente difficile da fare;
- facile da costruire difficile da rilevare.

2. **Dynamic Graph watermark** → consiste nell'andare ad incorporare il Watermark nella topologia di un grafo generato con alcuni input. Un algoritmo di Watermark di questo tipo è il **Path-Based Watermarking** → quest'ultimo inserisce una firma nella storia dei branch, che vengono scelti con un determinato input. Ovvero, a seconda che le guardie valutino a vero oppure a falso, genero un flusso di zeri e di 1. Dato, quindi, una sequenza di input, verrà scelto un particolare path ed in tale path verranno scelti dei punti di branch responsabili della codifica del Watermark e collezionando gli zeri e gli 1 degli IF posso estrarre la firma.



Questo algoritmo non è resistente, in quanto l'attaccante può inserire facilmente dei nuovi branch e di conseguenza non è resistente all'edge-flip. Per renderlo maggiormente resistente, si può rendere ridondante l'informazione che viene inserita.

3. **Dynamic Execution watermark** → consiste nell'incorporare il Watermark nella struttura della traccia. Un esempio è l'algoritmo di **Abstract Watermarking** → è un algoritmo che si trova a metà strada tra statico e dinamico, in quanto per estrarre il Watermark devo utilizzare un'analisi statica su un qualche dominio astratto, che di fatto (il dominio astratto) rappresenta la chiave per estrarre il Watermark. Tale algoritmo si basa sul **Teorema cinese del resto**, il quale consiste in: andare a prendere *n* numeri co-primi tra loro e li moltiplico tutti insieme. Il risultato della moltiplicazione (che chiamiamo *N*) ci indica, che possiamo codificare tutti i valori più piccoli di *N* (dato che i numeri co-primi sono infiniti, possiamo rendere N grande quanto vogliamo, in modo tale da codificare tutti i valori di cui necessitiamo). Dopodiché, il teorema cinese del resto si basa

sulle proprietà dei moduli, ovvero: va a calcolare il resto della divisione per ogni numero primo utilizzato e dati i resti e i numeri co-primi, siamo in grado di estrarre la firma.

L’ultima forma di Watermarking prende il nome di **Birthmark**, in cui di fatto viene eliminata la fase di embedding e di conseguenza vi è solamente la fase di estrazione → l’idea alla base del Birthmark, è che nel codice c’è qualcosa che sarà presente in ogni altra versione che possiamo creare del codice. Vi è, quindi, un “segno particolare” (come per esempio, una struttura dati o un algoritmo) che non si può andare a togliere. Possiamo, quindi, dire che manca la fase di embedding, perchè il mark non viene inserito dall’esterno, bensì è qualcosa che il codice contiene al suo interno. Gli algoritmi di Birthmark, quindi, cercano delle proprietà (che solitamente sono una combinazione di semantica e sintassi).