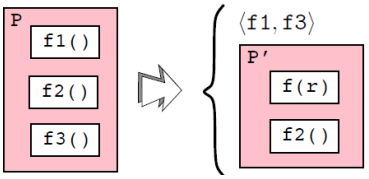
Similarity Analysis and Security

L’analisi di similarità ha tante applicazioni in ambito della sicurezza, per esempio viene utilizzata per:

- la clone detection, ovvero quando si vanno ad identificare i cloni all’interno del codice (quindi, quando viene scoperta una vulnerabilità nel codice, il nostro obiettivo è di andare a sistemarla ogni volta che la vulnerabilità si presenta nel sistema) → la clone detection è utile nella normale evoluzione del codice, per trovare delle occorrenze ripetute di una stessa funzionalità, le quali tipicamente vengono raggruppate in una chiamata a funzione, la quale (ovverosia la chiamata a funzione) avrà parametri diversi a seconda dei punti in cui verrà chiamata nel codice, per sostituire le occorrenze. Vediamo un esempio:



Nel programma *P*, le funzioni *f1* e *f3* vengono riconosciute come simili e di conseguenza, nel programma *P’* esse vengono sostituite con una chiamata a funzione *f*, a cui verranno passati dei parametri per specializzarla in *f1* oppure in *f3*. Da notare, che gli algoritmi di Clone Detection assumono che le funzioni, che possono essere riconosciute come simili (nel nostro esempio, *f1* e *f3*) **non siano sintatticamente identiche**, ma siano comunque molto simili a causa di aggiornamenti e/o personalizzazione del codice. In particolare, gli algoritmi di clone detection devono decidere una rappresenta conveniente del codice (come ad esempio, il control flow graph, oppure l’abstract syntax tree) e divide il codice in parti secondo un qualche criterio (dove queste parti potrebbero essere: le funzioni, le procedure o i moduli) e dopo di che si misura la similarità tra le varie parti di codice. Se la somiglianza è maggiore di una certa soglia, le parti verranno identificate come cloni e per ogni clone identificato, esso viene sostituito con una funzione, che viene opportunamente specializzata a seconda dei cloni sostituiti.

- riconoscere del codice plagiato, ovvero vogliamo andare a verificare che il codice distribuito, senza permesso, è simile (e quindi, di fatto, si sta facendo un’analisi di similarità) al mio codice originale;

- effettuare la software forensics → l'obiettivo principale della software forensics è rispondere alla domanda: "**Chi ha scritto il codice?**", quindi l'obiettivo è identificare l'autore di codice:
  - sia malevolo;
  - sia di copie illecite.

  Per riuscire a fare ciò, vengono utilizzati degli algoritmi di code attribution, che sono effettivamente degli algoritmi di similarità e che si basano sullo stile di programmazione. Gli algoritmi di code attribution funzionano molto bene sul codice ad alto livello, ma in realtà funzionano anche sul codice compilato. In particolare, per riuscire a fare code attribution dobbiamo avere a disposizione una serie di **samples** (ovvero di esempi di codice) per ogni autore e successivamente si dovranno fare delle analisi su tali sample, allo scopo di estrarre delle **features sintattiche** (quali ad esempio: la lunghezza media dei nodi delle variabili, i nomi delle funzioni, la spaziatura) che siano:
  - altamente variabili al variare dell'autore;
  - estremamente simili in programmi scritti dallo stesso autore.

  In questo modo, si riesce ad estrapolare l'autore del codice. La Software Forensics studia anche la:
  - **discriminazione del software** → la Software Forensics cerca di identificare quali parti un programma sono state scritte da un determinato autore e quali parti sono state scritte da autori diversi;
  - **caratterizzazione del software** → la Software Forensics cerca di trarre delle conclusioni sulla psicologia e sul background culturale ed educativo dell'autore del programma, in base agli aspetti stilistici di programmazione.
- effettuare code attribution → in questo caso particolare, l'analisi di similarità viene effettuata sugli stili di programmazione (mentre in altri casi, l'analisi di similarità si concentra maggiormente sugli aspetti sintattici del codice);
- gli algoritmi di birthmark → essi vanno in un qualche modo ad effettuare un'analisi di similarità, in quanto vanno a riconoscere una caratteristica comune al codice ottenuto modificando un certo codice proprietario. Ovvero, gli algoritmi di birthmark riconoscono il fatto, che un codice è in un qualche modo un'evoluzione (o comunque una modifica) di un altro codice, dato che contiene ancora il birthmark (ovvero il marchio distintivo) del programma originale. Chiaramente, questo birthmark del programma originale deve essere resistente

alle tecniche di offuscamento.
Ricordiamoci, infine, che gli algoritmi di birthmark sono degli algoritmi di Watermarking senza l'algoritmo di embedding.

- gli algoritmi di malware detection.

Vi sono diversi modi per decidere se due frammenti di codice sono simili oppure no. Abbiamo già detto, che tipicamente si vanno ad estrarre delle **features** dal codice e successivamente si vanno a confrontare tra loro. Immaginiamoci, quindi, di avere dei vettori di dimensione costante di features e assegniamo un valore a ciascuna di esse (quindi, il numero di features è sempre costante e cambia il loro valore). Da notare, che abbiamo diversi vettori di features, perchè:

- un vettore di features è per il programma che si vuole analizzare;
- un vettore di features per ogni sample utilizzato.

Una volta capito ciò, abbiamo che i modi per decidere se due frammenti di codice sono simili sono:

- **Hamming distance** → va a contare quante features sono uguali e quante features sono diverse. Quindi, due frammenti di codice sono simili quando hanno le stesse features e di conseguenza:
  - quando il numero di features uguali è molto grande, allora i due codici sono molto simili → il numero di features uguali viene poi rapportato con la dimensione del vettore, in modo tale da ottenere un "valore di similarità";
  - quando il numero di features diverse è molto grande, allora i due codici sono molto diversi → il numero di features diverse viene poi rapportato con la dimensione del vettore.
- **Edit distance** → dati i due frammenti di codice di cui voglio calcolare la similarità, l'edit distance conta quante operazioni di edit (ovvero quante modifiche) si devono compiere per trasformare il primo frammento di codice nel secondo frammento;
- **Quando le features sono degli insiemi**, possiamo utilizzare:
  - la **similarity** → controlla quante features hanno in comune i due vettori e tale valore viene rapportato con il numero totale delle features. In maniera formale, possiamo scrivere la similarity come:

$$|f(p) \cap f(q)|/(f(p) \cup f(q))$$

- il **containment** → controlla quante features hanno in comune i due vettori e tale valore viene rapportato con il numero di features del vettore del primo frammento di codice. In maniera formale, possiamo scrivere il containment come:

$$|f(p) \cap f(q)|/(f(p)$$

- Esistono, poi, una serie di misure di similarità fatte sui **grafi** → queste misure di similarità fatte sui grafi sono interessanti, in quanto i programmi possono essere rappresentati in vario modo attraverso i grafi. Sui grafi ci si rifà al concetto di **massimo sotto-grafo comune**, ovvero: Dati due grafi, voglio capire qual è il più grande sotto-grafo contenuto in tutti e due i grafi. Una volta ottenuto il massimo sotto-grafo comune, posso nuovamente calcolare le funzioni di similarity e containment, che in questo caso hanno la seguente forma:

$$similarity(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{\max(|G_1|, |G_2|)} \qquad containment(G_1, G_2) = \frac{|mcs(G_1, G_2)|}{|G_1|}.$$

Una tecnica utilizzata moltissimo per misurare la similarità è quella dei **K-grammi** → I K-grammi vengono utilizzati in molti ambiti, quali ad esempio:

- negli algoritmi di plagiarism detection;
- nell'analisi dell'authorship;
- negli algoritmi di birthmark detection.

Per capire cosa sono effettivamente i K-grammi, immaginiamoci di avere una stringa e di fissare un valore *K* (come per esempio, assegniamo K = 3). **I K-grammi sono tutte le sotto-stringhe contigue, all'interno della stringa originale, di lunghezza pari a *K*** (che nel nostro esempio, K = 3). Vediamo un esempio pratico:

y	a	b	b	a	d	a	b	b	a	d	o	O
1	2	3	4	5	6	7	8	9	10	11	12	13

by considering a window of size 3 over A we get the following set of 3-grams of A:

`yab abb bba bad ada dab abb bba bad ado doo`

Abbiamo la nostra stringa iniziale e fissiamo il valore di K = 3. In questo modo, abbiamo che tutti i possibili K-grammi sono: yab, abb, bad, ….

💡 A questo punto, per misurare la similarità vado a controllare quanti K-grammi hanno in comune i due programmi → **possiamo chiaramente capire, che l'elemento fondamentale dei K-grammi è il valore che assegniamo alla variabile K**, ovvero deve avere un valore sufficientemente grande per catturare tutte le parole chiave del linguaggio. Inoltre, dobbiamo anche dire il fatto, che **i K-grammi sono resistenti alle permutazioni**.

Per rendere più efficiente l'utilizzo dei K-grammi vengono utilizzate le hash, ovvero invece di memorizzare tutti i K-grammi, potrebbe essere utile memorizzare le hash e di conseguenza dovremmo avere una tabella, che mi permette di far corrispondere le hash ai rispettivi K-grammi:

hash(obe) = 15	hash(abb) = 2	hash(bee) = 16
hash(bba) = 3	hash(bad) = 4	hash(yab) = 8
hash(ydo) = 14	hash(eed) = 17	hash(byd) = 13
hash(doo) = 1	hash(ada) = 5	hash(edo) = 18
hash(ado) = 7	hash(coo) = 10	hash(dab) = 6
hash(oob) = 11	hash(oby) = 12	hash(sco) = 9

Una volta realizzata la tabella, andiamo a prendere la stringa iniziale e otteniamo la seguente tabella:

yab	abb	bba	bad	ada	dab	abb	bba	bad	ado	doo
$A_0:8$	$A_1:2$	$A_2:3$	$A_3:4$	$A_4:5$	$A_5:6$	$A_6:2$	$A_7:3$	$A_8:4$	$A_9:7$	$A_{10}:1$

Possiamo vedere, ad esempio, che "yab" ha come hash il valore 8 della tabella precedente e così con gli altri.

Nell'ipotesi in cui i testi fossero molto lunghi, possiamo anche decidere di non guardare tutti i K-grammi, ma ne prendiamo uno ogni tanto, in particolare ogni K-grammo in posizione *0 mod p* (zero modulo *p*), dove *p* è un valore che decidiamo noi → scegliendo i K-grammi in questo modo, però, si è notato che essi sono troppo lontani tra loro e di conseguenza si è deciso di utilizzare un algoritmo di similarità chiamato **MOSS**. Esso va a eseguire una finestra di dimensione W sulle sequenze di hash e **va a mantenere l'hash più piccolo in ogni finestra**. Per capire meglio il funzionamento del MOSS riprendiamo l'esempio di prima:

$(A_0:8$	$A_1:2$	$A_2:3$	$A_3:4)$
$(A_1:2$	$A_2:3$	$A_3:4$	$A_4:5)$
$(A_2:3$	$A_3:4$	$A_4:5$	$A_5:6)$
$(A_3:4$	$A_4:5$	$A_5:6$	$A_6:2)$
$(A_4:5$	$A_5:6$	$A_6:2$	$A_7:3)$
$(A_5:6$	$A_6:2$	$A_7:3$	$A_8:4)$
$(A_6:2$	$A_7:3$	$A_8:4$	$A_9:7)$
$(A_7:3$	$A_8:4$	$A_9:7$	$A_{10}:1)$

Vediamo, che abbiamo una finestra di dimensione 4 e per ogni finestra, andiamo a mantenere l'hash di dimensione inferiore.

In questo modo abbiamo una maggiore efficienza, in quanto **invece di mantenere le hash di tutti i K-grammi, vengono mantenute solamente le hash di dimensione inferiori di ogni finestra**.

A questo punto, supponiamo di aver eseguito l'algoritmo di MOSS su tre documenti diversi (A,B e C), ottenendo le seguenti hash (dei K-grammi):

- document A containing the string yabbadabbadoo with hashes $\{A_1:2, A_2:3, A_6:2, A_{10}:1\}$
- document B containing the string scoobydoobydoo with hashes $\{B_0:9, B_1:10, B_2:11, B_6:1, B_7:11, B_{11}:1\}$
- document C containing the string doobeedoobeedoo with hashes $\{C_0:1, C_1:11, C_2:15, C_6:1, C_7:11, C_8:15, C_{12}:1\}$

Vogliamo, quindi, confrontare i K-grammi ottenuti e per fare questo utilizziamo una tabella, che ha:

- sulla prima colonna, i valori di hash che abbiamo mantenuto nei nostri documenti;
- per ogni valore di hash, indico in **quale** documento compare e **quante volte** compare nel documento.

1	$A_{10}:1$	$B_6:1$	$B_{11}:1$	$C_0:1$	$C_6:1$	$C_{12}:1$
2	$A_1:2$	$A_6:2$				
3	$A_2:3$					
9	$B_0:9$					
10	$B_1:10$					
11	$B_2:11$	$B_7:11$	$C_1:11$	$C_7:11$		
15	$C_2:15$	$C_8:15$				

Per esempio l'hash 1 compare una volta in A, 2 volte in B e 3 volte in C. L'hash 2, invece, compare 2 volte in A e così via.

Successivamente, costruisco una tabella, la quale per ogni documento colleziona le hash di ogni specifico documento, che sono presenti anche negli altri due documenti:

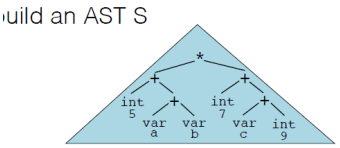
A	$B_6:1$	$B_{11}:1$	$C_0:1$	$C_6:1$	$C_{12}:1$	
B	$A_{10}:1$	$C_0:1$	$C_6:1$	$C_{12}:1$	$C_1:11$	$C_7:11$
C	$A_{10}:1$	$B_6:1$	$B_{11}:1$	$B_2:11$	$B_7:11$	

Infine, per ogni coppia di documenti, dobbiamo prendere il loro elenco di hash ed estrarre quelle che hanno in comune e chiaramente, le coppie di documenti con l'elenco più lungo di hash, sono con ogni probabilità casi di plagio:

[A,B]	$B_6:1$	$B_{11}:1$	$A_{10}:1$						
[A,C]	$A_{10}:1$	$C_0:1$	$C_6:1$	$C_{12}:1$					
[B,C]	$C_0:1$	$C_6:1$	$C_{12}:1$	$C_1:11$	$C_7:11$	$B_6:1$	$B_{11}:1$	$B_2:11$	$B_7:11$

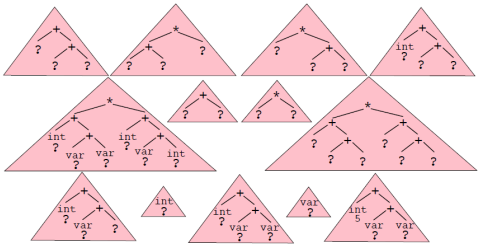
Ci sono, poi, degli **algoritmi di similarità che si basano sugli Abstract Syntax Tree**, dato che abbiamo detto che gli Abstract Syntax Tree sono presenti nei nostri programmi e di conseguenza, potrebbe essere utile capire quanto sono simili tali strutture sintattiche. Vediamo un esempio di Abstract Syntax Tree su un'espressione:

```
(5 + (a + b)) * (7 + (c + 9))
```



Vediamo che abbiamo in cima gli operatori e sulle foglie abbiamo gli operandi.

Una volta ottenuti gli Abstract Syntax Tree vengono creati dei **pattern** sugli alberi. In particolare, i pattern sono dei sotto-alberi di un Abstract Syntax Tree, in cui uno o più sotto-alberi sono stati sostituiti da un carattere jolly. Vediamo un esempio:



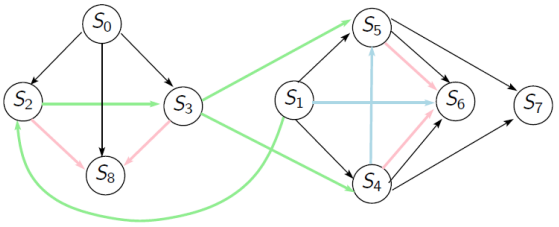
In alto a sinistra, abbiamo un pattern che mi indica che andremo a sommare qualcosa alla somma di altre due cose. Il pattern, quindi, mi specifica una struttura (non me la specifica nel dettaglio). In particolare, un pattern matcha un Abstract Syntax Tree, se il pattern può essere specializzato per ottenere effettivamente l'Abstract Syntax Tree.

Vi sono, poi, degli algoritmi di similarità che si basano sui grafi e sono molto interessanti, in quanto diversi aspetti di un programma possono essere rappresentati come dei grafi. Il problema di trovare il sotto-grafo comune è un problema NP, però vi sono delle euristiche che funzionano bene per i grafi legati ai programmi. Per esempio, ci sono algoritmi che cercano il sotto-grafo comune partendo dai Problem Dependencies Graph, ovvero partendo da un grafo in cui ci sono:

- i **nodi**, che sono dati dagli statement;
- gli **edge**, che catturano sia le dipendenze di flusso sia le dipendenze di controllo (le dipendenze di controllo legano l'esecuzione del programma, ovvero stabiliscono quale istruzione viene eseguita prima di un'altra, oppure legano le guardie a tutti gli statements presenti nel ciclo o nei branch).

Prendiamo, allora, un programma e costruiamo il suo Problem Dependencies Graph, che è costituito dai nodi (che sono tutti gli statement) e sono legati dagli archi (che sono le dipendenze di flusso e di controllo):

```
S0 : int k = 0;
S1 : int s = 1;
S2 : while (k < w) {
S3 :    if (x[k] == 1)
S4 :        R = (s*y) % n;
     else
S5 :        R = s;
S6 :    s = R*R % n;
S7 :    L = R;
S8 :    k=k+1;
   }
```



Quindi, per fare Clone Detection bisognerebbe andare a costruire il Problem Dependencies Graph e poi si dovrebbero identificare i sotto-grafi comuni, i quali avranno:

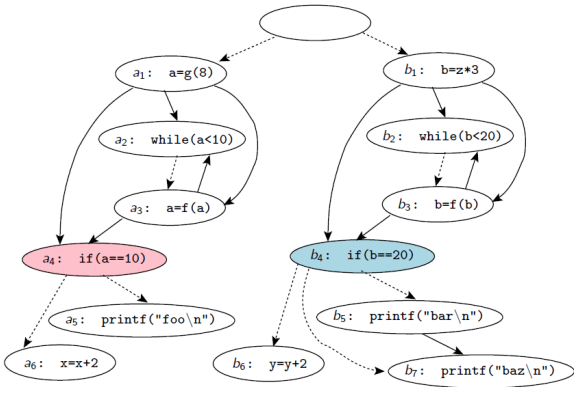
- la **stessa shape**, ovvero avranno i nodi nelle stesse posizioni;

- i **nodi dovranno matchare**, ovverosia i nodi dovranno avere la stessa struttura sintattica.

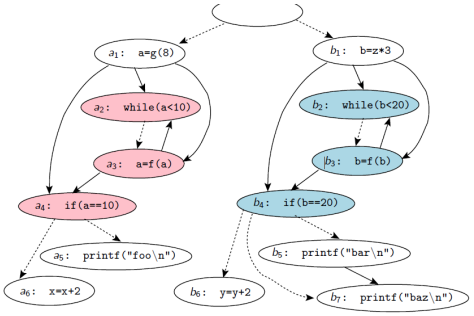
Vediamo un esempio di ciò:

```
a1: a = g(8);
b1: b = z*3;
a2: while(a<10)
    a3: a = f(a);
b2: while(b<20)
    b3: b = f(b);
a4: if (a==10) {
    a5: printf("foo\n");
    a6: x=x+2;
}
b4: if (b==20) {
    b5: printf("bar\n");
    b6: y=y+2;
    b7: printf("baz\n");
}
```

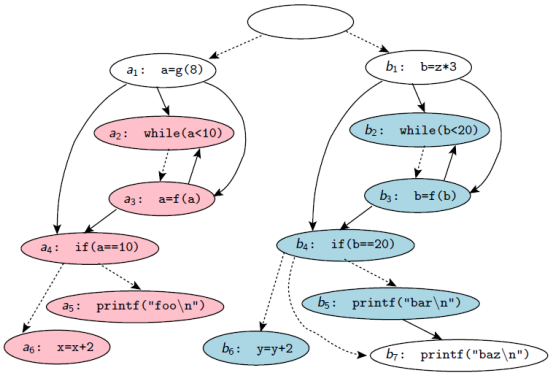
Abbiamo due frammenti di codice, che sono stati alternati (abbiamo, cioè, uno statement del codice di *A* e uno statement del codice di *B*) e il nostro obiettivo, è capire se i due frammenti di codice *A* e *B*, sono simili tra loro. Vado, allora, a costruire il Problem Dependencies Graph e possiamo vedere immediatamente, che gli statement dei due frammenti di codice vengono ordinati:



Una volta costruito il Problem Dependencies Graph devo andare a prendere due statement che matchano sintatticamente (in questo caso prendo: *if(a==10)* e *if(b==20),* dato che abbiamo in entrambi i casi un confronto fra una variabile e un intero) e poi procedendo all'indietro, vado a considerare gli statement che matchano, ovvero gli statement che hanno la stessa struttura sintattica.



Una volta che ho finito di controllare se all'indietro vi sono degli statement che matchano, devo andare a controllare in avanti e di conseguenza ottengo:



A questo punto ho ricavato i due sotto-grafi comuni e di conseguenza, a questo punto, posso estrarre la funzione CLONE, la quale viene opportunamente istanziata per i due sotto-grafi.

Tutti gli algoritmi visti fino a questo momento lavorano su qualsiasi tipologia di codice (nel senso, che non ci siamo preoccupati se stessimo analizzando codice sorgente, codice compilato oppure codice ad alto livello), dato che lavoravano sulle rappresentazioni astratte del codice (quali ad esempio: Control Flow graph oppure gli Abstract Syntax Tree). Vi sono, poi, degli algoritmi di similarità sviluppati per il codice binario, in quanto nel codice binario la similarità è utile per due grandi motivi, ovvero:

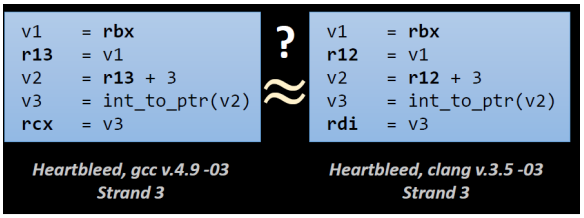
1. per trovare le **vulnerabilità** → per riuscire a fare ciò, è stato sviluppato un algoritmo di similarità chiamato **BinDiff** → esso si basa sul Control Flow Graph e sul Call graph, quindi fa un'analisi di similarità sui grafi. Successivamente a BinDiff è stato sviluppato **BinHunt**, il quale era una versione più "evoluta" di BinDiff, in quanto si concentrava maggiormente sugli aspetti semantici del codice binario e utilizzava delle tecniche di similarità più evolute. Successivamente a BinHunt è stato sviluppato **BinJuice**, il quale voleva diventare ancora più semantico e l'idea alla base di BinJuice era quella di estrarre il juice (ovvero il succo) del codice assembly, ovvero: dal codice assembly, BinJuice estraeva la

semantica e dopo di ché veniva generalizzata la semantica (questa generalizzazione prende il nome di **juice**).

2. per fare **malware detection**.

Statistical Similarity of Binaries

Parliamo, infine, di questo lavoro che fa analisi di similarità sui codici binari, al fine di individuare le vulnerabilità. Questo lavoro parte dal presupposto che: “**Due cose sono simili se sono fatte da pezzi simili**” → questa idea viene molto utilizzata nell’analisi delle immagini. Infatti, si dice che un'immagine è simile a un'altra, se può essere composta utilizzando regioni dell'altra immagine. Questa somiglianza può essere quantificata utilizzando un ragionamento statistico. L’idea di questo lavoro è di applicare lo stesso concetto al codice, ovvero: “**Possiamo dire che due codici binari sono simili, perchè sono fatti da pezzi simili?**” Vanno, allora, a prendere il codice binario e lavorano a livello di Basic Block e si chiedono: “la query *q* (ovvero i samples) è simile al target *t*?” Per ogni Basic Block estraggono gli Strands (ovvero i filamenti) → lo strand è l'insieme delle istruzioni necessarie per calcolare il valore di una determinata variabile all’interno di un Basic Block. Quindi, abbiamo che ogni Basic Block viene suddivido in strands, che dicono cosa calcola ogni Basic Block per ogni variabile del Basic Block (ovvero: viene suddivido un Basic Block in strands, ognuno dei quali mi indica cosa calcola, per ogni variabile (del Basic Block), il Basic Block). Chiaramente, due Basic Block sono simili se sono composti da strands simili (ovvero se hanno molti strands in comune) e successivamente attraverso **un’analisi statistica** vanno a verificare se due codici binari sono simili. Sorge spontanea la domanda: “**Quando due strands sono simili?**” Due strands matchano, quando le variabili di uno strand matchano con le variabili del secondo strand e per verificare ciò, essendo che gli strands sono molto piccoli, vanno a provare tutte le variabili dei due strands.



YES!