**Multiple interpretari ale unei expresii regulate in diverse contexte**

**ABSTRACT**

In ziua de azi expresiile regulare devin din ce in ce mai des folosite si din ce in ce mai multi oameni incep sa invete sa le foloseasca. Mai multi oameni rezulta mai multe stagii de pregatire si intelegere diferite. Din cauza pregatirii precare sau a studiului insuficient, pe piata pot aparea regex-uri vagi care pot compromite integritatea unei aplicatii sau serviciu. Astfel de interpretari sau regex-uri vagi pot conduce de la functionarea incorecta a unei aplicatii sau servciu pana la vulnerabilitati de sistem precum ReDOS care pot avea efecte negative pe termen lung din punct de vedere financiar.

Numarul de regex-uri vagi/problematice de pe piata nu se stie cu exactitate deoarece unele regex-uri pot fi vagi pentru anumite Regex Engine-uri sau corecte pentru altele.

Acesta lucrare analizeaza modul in care sunt prelucrate si interpretate regex-urile pentru a putea determina cum apar astfel de posibile interpretari, explica cum se produc unele efecte negative ale folosirii de regex-uri vagi si care sunt recomandarile pentru a putea evita aceste efecte.

ACM: I.2.7: Natural Language Processing

AMS: 68T50: Natural Language Processing

1. **INTRODUCERE**

**Ce sunt Regex-urile?**

Regex sau expresie regulara (regular expression) este un sir de caractere sub forma unui sablon care pus deasupra unui text poate gasi anumite secvente din text care respecta acel sablon.

**Unde sunt folosite Regex-urile?**

Regex-urile pot fi gasite in foarte multe zone de activitate, de la comenzi UNIX de procesare de text (grep,sed,awk) si analizatoare lexicale (folosite in compilatoare si interpretoare) pana la motoare de cautare si aplicatii care se ocupa de procesarea si editarea de documente text (Microsoft Word, Notepad, IDE-uri).

**Cum sunt folosite Regex-urile?**

Desi un regex nu are o forma foarte naturala si in functie de domeniul problemei in care sunt folosite acestea dimensiunea si complexitatea unui regex poate creste semnificativ de mult, ele reprezinta un mod foarte rapid si eficient de a cauta anumite cuvinte cheie sau structuri intr-un text.

Apare urmatoarea intrebare: Cum sunt implementate Regex-urile? Pentru a putea profita de facilitatile oferite de expresiile regulare este nevoie de un Regex Engine. La ora actuala exista mai multe tipuri de Regex Engine-uri care difera in ceea ce priveste partea de preprocesare a regex-urilor si de procesare a textului de intrare, dar care in final ar trebui sa aibe acelasi efect. Vom vedea in capitolele urmatoare ca in unele cazuri nu este chiar adevarat, adica un regex X cu un text de intrare Y poate avea mai multe iesiri sau rezultate.

**Ce urmeaza?**

In Cap. 2 vom explica mai in detaliu ce sunt Regex Engine-urile,cum functioneaza acestea si care sunt principalele strategii de potrivire a regex-urilor.

In Cap. 3 vom folosi notiunile discutate in capitolul precedent pentru a determina care sunt problemele care pot aparea in urma folosirii incorecte a regex-urilor cu demonstratii si exemple.

Tot in acest capitol vom incerca sa raspundem si la urmatoarea intrebare : De ce expresiile regulare pot avea multiple interpretari in functie de context/platforme?

In Cap. 4 vom discuta care sunt alternative/recomandari pentru a preveni problemele enuntate mai sus si cum pot fi aplicate.  
 In Cap. 5 vom concluziona toate cele discutate mai sus si vom enunta ce vom face in continuare.

1. **REGEX ENGINES SI STRATEGII DE POTRIVIRE**

Vom incepe prin a defini ce este un Regex Engine.

Un Regex Engine este o entitate sau un fel de analizator lexical care poate transforma/traduce o expresie regulara sub o alta forma care poate fi folosita mai departe pentru a parcurge un text sursa si pentru a putea descoperi anumite structuri din acesta.

Acesta primeste doua date de intrare, regex-ul ce trebuie procesat si textul sursa ce trebuie analizat, parcurgand fiecare data de intrare de la stanga la dreapta caracter cu caracter. De obicei un Regex Engine este implementat ca un automat finit, adica o masina care lucreaza cu stari.

**2.1 Procesarea Regex-urilor**

Pentru a putea intelege cum un Regex Engine proceseaza un regex, trebuie sa definim cateva notiuni.

Sa luam urmatorul regex “ab\*c”. Acest regex va gasi toate secventele din text care incep cu “a”, pot avea 0 sau mai multi de “b” dupa si se termina cu “c”. Deci secvente precum “ac”, “abc” sau “abbbbc” sunt acceptate de catre acea expresie regulata. Alte secvente precum “a” sau “adc” sau “bc” nu sunt acceptate de catre acel regex. Cum majoritatea implementarilor Regex Engine-urilor au la baza Teoria Automatului Finit, vom incerca sa reprezentam regex-ul de mai sus sub forma unui grafic cu stari cum este in figura 2.

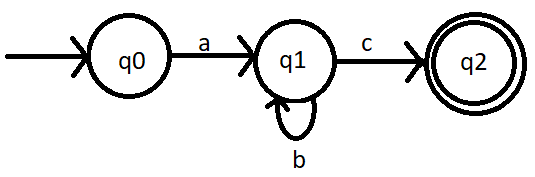
****

Figura 2 - reprezentarea grafica a unui automat pentru “ab\*c”.

Daca analizam figura 2, automatul nostru are 3 stari, unde q0 este starea initiala si q2 este starea finala, si 3 tranzitii, de la q0 la q1 prin “a”, de la q1 la q1 prin “b” si de la q1 la q2 prin “c”. Reprezentarea este similara cu definitia regex-ului. Regexul “ab\*c” presupune o secventa care incepe cu “a”, poate fi urmata de oricati de “b” si se termina cu un “c”, adica este similara cu tranzitiile de mai sus.

Luam urmatoarea secventa de intrare “abc”. Pornim din starea initiala adica din q0, parcurgem secventa si luam primul caracter, adica “a”, din q0 daca o luam prin “a” ajungem in q1. Continuam sa parcurgem secventa si ajungem la “b” cu starea curenta in q1, din q1 daca o luam prin “b” ajungem tot in q1. Continuam cu secventa ramasa si ajungem la “c” cu starea curenta in q1, din q1 daca o luam prin “c” ajungem la q2 care este o stare finala, deci regex-ul “ab\*c” accepta secventa “abc”.

Luam urmatoarea secventa “aa”. Pronit la fel din starea initiala q0 si luam primul “a”, din q0 prin “a” ajungem in q1. Continuam si luam urmatorul “a” cu starea curenta in q1, din q1 nu putem ajunge nicaieri prin “a” pentru ca nu exista un drum prin “a”. Deci ne oprim in q1 care nu este o stare finala precum q2, deci regex-ul “ab\*c” nu accepta secventa “aa”.

Un astfel de automat ca cel din figura 2 este un automat finit determinist (AFD) deoarece are un numar finit de stari si in functie de secventa de intrare se poate deduce care este flow-ul de executie.

Mai exista si notiunea de automat finit nedeterminist (AFN) care are un numar finit de stari dar care nu se poate deduce flow-ul de xecutie, adica dintr-o stare q prin tranzitia “a” se poate ajunge in mai multe stari. Un exemplu de de AFN este in figura 3.

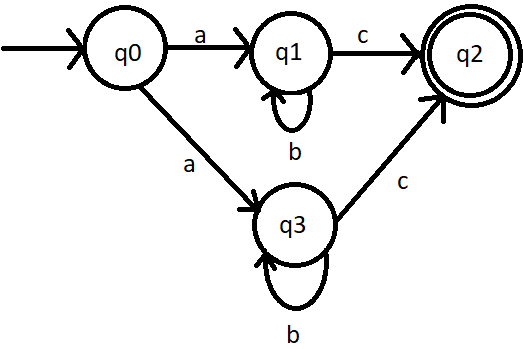


Figura 3 - Acesta poate fi un automat finit pentru regex-ul “ab\*c|ad\*c”. Acesta este un AFN deoarece din q0 prin “a” se poate ajunge in q1 si q3, deci pentru secventa de intrare “ac”, flow-ul de executie ar fi q0-q1-q2 sau q0-q3-q2. Ambele solutii sunt corecte, dar nu se poate deduce flow-ul de executie.

Exista o teorema conform careia orice AFN poate fi convertit intr-un AFD si viceversa. Spre exemplu, AFD-ul echivalent pentru figura 3 este figura 4.

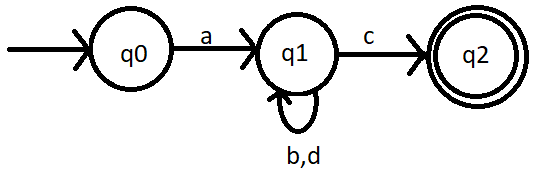


Figura 4 - AFD ul echivalent pentru AFN-ul din figura 3. Din q1 prin “b” se poate ajunge in q1, prin “d” tot in q1, deci se poate deduce flow-ul de executie.

Deci un Regex Engine cand proceseaza un regex, incearca sa transforme acel regex intr-un automat finit. De obicei automatul finit este un AFN pentru ca este mai compact si mai rapid, iar conversia lui catre un AFD ar necesita niste stari suplimentare deci mai multa memorie si timp de executie chiar daca AFD-ul este mai eficient si precis.

Eventual inainte de procesare se poate face o preprocesare in care metacaractere precum “\w” sau “\d” sunt inlocuite cu expresiile lor “[a-zA-Z]”, respectiv “[0-9]” si alte operatii.

Regex Engine-urile folosite de comenzile UNIX de procesare a textului precum grep,sed,awk sau alte limbaje precum lex, flex MySQL creeaza un AFD. Regex Engine-urile pentru limbajele de nivel inalt precum Java, C++, Python, PHP creeaza un AFN.

**2.2 Procesarea textului de intrare**

Am enuntat mai sus cum se proceseaza regex-urile, iar procesarea secventei de intrare ar presupune trecerea prin automatul finit si verificarea daca s-a ajuns in starea finala. Usor nu? Din pacate nu.

Unele expresii regulare precum “a.\*c” sunt prea “lacome” (greedy).

Secventa de mai sus se traduce “ orice secventa care incepe cu <<a>> care este urmata de oricate caractere si se termina cu <<c>>” si se reprezinta ca in figura 5.

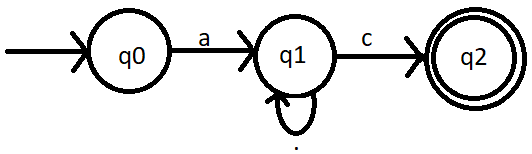


Figura 5 - Automatul finit nedeterminist pentru “a.\*c”.

Daca avem urmatoarea secventa de intrare “ac”. Din q0 mergem prin “a” ajungem in q1, din q1 teoretic sunt 2 stari in care se poate ajunge prin c, q1 si q2, asa ca se va considera oridinea din regex, adica din q1 se merge prin “.” (“c” -ul din secventa) si se va ajunge tot in q1. In aceasta situatie, secventa este prelucrata complet si nu s-a ajuns in starea finala, deci secventa “ac” nu este acceptata desi in mod normal ar trebui. In cazul de fata “.\*” a incercat sa potriveasca cea mai lunga secventa pe care o putea gasi ajungand astfel la capatul secventei.

Exista mai multe solutii la problema asta.

Prima solutie presupune sa luam toate cazurile posibile, adica sa trecem secventa de intrare prin tot automatul finit pe toate ramurile si sa determinam cea mai mare secventa potrivita. Adica sa analizam flow-ul q0-a-q1-.-q1 si q0-a-q1-c-q2 si vedem ca al doilea flow returneaza “ac” care este egala cu secventa initiala.

Astfel de Regex Engine-uri sunt Text-Directed, nu presupun pic de backtracking.

A doua solutie ar fi sa mergem inapoi si sa incercam o alta varianta, un alt flow (q0-a-q1-c-q2).Deci folosim backtracking care incearca sa potriveasca cat se poate de bine fara sa ia in considerare toate cazurile. Astfel de Regex Engine-uri sunt Regex-Directed si sunt cele mai folosite.

Alte secvente sunt prea “dornice”(eager) si cauta secventa de lungime minima pe care o pot potrivi. Spre exemplu “\w{2,3}?” care va potrivi secvente cu 2-3 litere. Pentru secventa “abc” se va potrivi doar “ab”, daca vrem musai sa se potriveasca “abc” putem forta lucrurile schimband regex-ul in “\w{2,3}?$”, acum va potrivi doar “abc”. Cele doua solutii de mai sus se aplica si aici.

Majoritatea Regex Engine-urilor care implementeaza un AFN folosesc algortimul lui Thomson pentru crearea AFN-urilor datorita complexitatii liniare fata de cea exponentiala care ar fi in mod normal, astfel se previn backtracking-urile inutile in partea de procesare a textului de intrarea prin crearea unui AFN cat mai AFD.

1. **CONSECINTE SI EFECTE ALE INTERPRETARII EXPRESIILOR REGULATE**

**3.1 Multiple interpretari**

In capitolul precedent am discutat despre cum sunt implementate Regex Engine-urile. Am discutat despre faptul ca majoritatea Regex Engine-urilor sunt implementate ca AFD sau AFN.

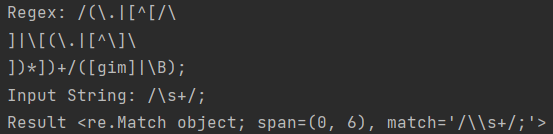
Regex Engine-urile care implementeaza AFN consuma mai putina memorie, au un timp de procesare mult mai mic si sunt mai simple, pe langa asta sunt mult mai volative si este mult mai usor de adaugat anumite imbunatatiri/functionalitati precum lazy quantifiers (cuantificatori lenesi) care pot forta un Regex Engine care e greedy sa fie mai eager.

Desi AFN-urile par mai bune, el au alte dezavantaje precum precizia mai slaba decat a AFD-urilor si controlul slab asupra backtracking-ului(daca este cazul).

Regex Engine-urile cu AFN pot avea mai multe iesiri/rezultate pentru acelasi regex si aceeasi secventa de intrare. Toate acestea sunt insa direct proportionale cu dimensiunea datelor de intrare, adica cu cat un regex este mai mare cu atat AFN-ul rezultate este mai complex si cu cat textul de intrare este mai mare cu atat timpul de executie este mai mare si rezultatele pot fi mai diverse.

Spre exemplu, limbajele de nivel inalt precum Java si Python folosesc Regex Engine-uri care implementeaza un AFN, dar implementari sunt diferite chiar daca au cam aceleasi efecte, deci pentru o expresie regulara lunga de 50 de caractere AFN-ul rezultat de Regex Engine-ul din Java este putin diferit fata de cel rezultat de Regex Engine-ul din Python. AFN-uri diferite poate rezulta in rezultate diferite in cazul unui text de intrare destul de mare.

Pe langa faptul ca implementarile sunt diferite, si sintaxa poate fi diferita, spre exemplu, daca folosim un regex care cauta si caraterul “[“ care este un token special in sintaxa regex (BRE), in Java acest caracter trebuie insotit de caracterul de evadare “\”, dar in Python nu este o obligatoriu. In figura 6 este demonstrat aces lucru.



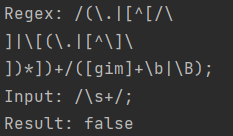


Figura 6 - Avem doua rulari in Python (prima imagine) si Java(a doua imagine) pentru regexul "/(\\.|[^[/\\\n]|\[(\\.|[^\]\\\n])\*])+/([gim]+\b|\B);" si secventa de intrare “/\s+/;”. In Python regex-ul si input-ul nu au fost prelucrate, faptul ca se returneaza un obiect de tip Match inseamna ca regex-ul potriveste secventa de intrare. In Java pentru a putea compila exemplu au trebuit adaugate niste caractere de evadare, dar chiar si asa, rezultatul false ne arata faptul ca regex-ul nu potriveste secventa de intrare.

In afara de tipul de implementare, strategiile diferite de potrivire a secventelor de intrare pot duce la rezultate diferite pentru acelasi regex si aceeasi secventa de intrare.

Spre exemplu Regex Engine-urile de tip Regex-Direct pot implementa functionalitati precum lazy quantifiers care prin adaugarea unui “?” in fata unui token precum “\*” sau “+” poate sa-i schimbe usor comportamentul, in sensul ca, in loc sa caute cea mai multa secventa pe care o poate potrivi, o va cauta pe cea mai scurta. Lazy quantifiers nu functioneaza in cazul Regex Engine-urilor de tip Text-Direct. Deci un regex care foloseste lazy quantifiers intr-un Regex-Direct va avea cel mai probabil un rezultat diferit intr-un Text-Direct daca regex-ul a fost creat tocmai cu scopul de a folosi lazy quantifiers.

**3.2 Consecinte**

Datorita interpretarii si procesarii multiple/ diferite a regex-urilor pot aparea probleme de integritate si compatibilitate.

Deoarece limbaje de nivel inalt folosesc Regex Engine-uri diferite, dupa cum am aratat si in figura 6, acelasi regex poate fi interpretat diferit, deci acelasi regex nu poate fi folosit in ambele contexte si trebuie facut cate unul specific pentru fiecare limbaj. Deci pot aparea probleme de compatibilitate la capitolul acesta intre doua aplicatii scrise in aceste doua limbaje. Plus ca mai apare intrebarea, care Regex Engine este mai bun? Din pacate nu se stie raspunsul la acestea intrebare, depinde foarte multe de context. Pentru regex-uri mici, Regex Engine-urile se comporta la fel, dar cand regex-ul devine mai complex, unele devin mai rapide, alte mai precise.

De obicei programele care folosesc ElasticSearch se pot confrunta cateodata cu problema asta.

Pe langa problema de integritate si de compatibilitate mai poate aparea si o problema de secutitate.

Pentru a preveni atacurile de tip SQL Injectiun, XSS, CSRF, etc… aplicatiile web filtreaza si valideaza input-ul de la utilizator (input sanitize), iar cel mai rapid mod de a face asta e prin folosirea regex-urilor.

Majoritatea Regex Engine-urilor folosesc backtracking in partea de potrivire a secventei de intrare. Astfel pentru un text de intrare suficicient de mare, complex si ambiguu o simpla procesare poate dura de la cateva minute pana la cateva ore. Folosing aceeasi tehnica pe mai multe conexiuni, reteaua se poate aglomera destul de rapid si serviciul poate deveni inaccesibil. Un astfel de atac poarta denumirea de ReDOS (Regular Expression Denial of Service) attack. Un astfel de atac poate fi greu de depistat si poate crea foarte usor alte brense de securitate intrucat server-ul este mai preocupat de sanizarea input-urilor decat de propria siguranta.

1. **RECOMANDARI**

Majoritatea interpretarilor multiple sunt cauzate ori de implementarile diferite ale Regex Engine-urilor ori de regex-urile mult prea complexe.

La ora actuala cele mai folosite Regex Engine-uri sunt cele care implementeaza un AFN deoarece contin mai multe functionalitati precum lazy quantifiers, pentru a dezactiva modul greedy al Regex Engine-urilor; backreferences prin care se pot refolosi subsecvente din regex; atomic grouping care ofera un control mai mare asupra backtracking-ului. Toate aceste functionalitati au rolul de a mari precizia unui Regex Engine pentru a evita interpretarile multiple si pentru a oferi un mod mai usor de a lucra cu regex-urile.

Bineinteles ca depinde foarte mult de contextul in care trebuie folosit un Regex Engine, pentru rapiditate se foloseste AFN, pentru precizie maxima se foloseste AFD, pentru ambele se folosex Regex Engine-urile hibride care contruiesc un AFN si il convertesc intr-un AFD daca este cazul.

Pe langa diferitele implementari, o alta cauza a interpretarii multiple este competenta scazuta sau lipsa a celui care scrie regex-urile. Regex-urile prea lungi duc la automate finite foarte complexe, care pot duce la rezultate diferite. Folosirea de functionalitati, spre exemplu lazy quantifiers in Regex Engine-urile de tip Text-Directed poate sa nu aibe efectul dorit. Regex-urile vagi sau cele de genul “a.\*.\*.\*.\*.\*c” care presupun un calcul computational destul de mare si un backtracking pe masura poate sa duca ori la un rezultat incorect ori la oprirea procesarii textului de intrare (stiva de executie este prea mare din cauza backtracking-ului).

Rgex-urile simple si scrise corect sunt cele mai de incredere si cele mai predictibile. Refactorizarea regex-urilor poate rezolva multe probleme. Adaptarea regex-urilor in functie de Regex Engine si de limbaj poate fi o solutie pentru a evidat problemele de compatibilitate.

Limitarea textului de intrare poate preveni procesarea indelungata a regex-urilor prevenind astfel atacurile de tip ReDOS.

1. **CONCLUZII**

In concluzie, regex-urile ne ofera o modalitate rapida si eficienta de a procesa un text. Acestea au nevoie de un Regex Engine pentru a putea fi executate si in functie de context, scop si implementarea Regex Engine-urilor acestea pot sa aibe rezultate multiple.

Astfel de multiple interpretari pot cauza functionarea defectuoasa a unui serviciu/aplicatii sau pot duce la vulnerabilitati precum ReDOS. Astfel de probleme sunt greu de depistat intrucat nu se poate face debugging pe regex-uri. Cea mai utilizata metoda de a verifica corectitudinea unui regex sunt rularea pe anumite platforme precum regex101.com .

Si competenta celui care scrie regex-urile conteaza foarte mult. Expresiile regulare vagi sau incomplete pot cauza efecte nedorite si probleme care sunt greu de detectat.

De obicei studiul aprofundat al documentatiilor, mitigarea vulnerabilitatilor si profesionalismul pot rezolva astfel de probleme.

1. **REFERINTE**

* Xi Ye, Qiaochu Chen, Xinyu Wang, Isil Dillig and Greg Durrett (2020). Sketch-Driven Regular Expression Generation from Natural Language and Examples.
* Ye, X., Chen, Q., Dillig, I., & Durrett, G. (2020). Benchmarking Multimodal Regex Synthesis with Complex Structures. ArXiv, abs/2005.00663.
* Prakash M Nadkarni, Lucila Ohno-Machado, Wendy W Chapman . Natural language processing: an introduction .
* Jun-U Park, Sang-Ki Ko, Marco Cognetta, and Yo-Sub Han. 2019. SoftRegex: Generating Regex from Natural Language Descriptions using Softened Regex Equivalence. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 6425–6431, Hong Kong, China. Association for Computational Linguistics
* Nate Kushman & Regina Barzilay. Using Semantic Unification to Generate Regular Expressions from Natural Language.
* D. Regéciová, D. Kolář and M. Milkovič, "Pattern Matching in YARA: Improved Aho-Corasick Algorithm," in IEEE Access, vol. 9, pp. 62857-62866, 2021, doi: 10.1109/ACCESS.2021.3074801.
* P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal and H. Noyes, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 12, pp. 3088-3098, Dec. 2014, doi: 10.1109/TPDS.2014.8.
* F. Yi, B. Jiang, L. Wang and J. Wu, "Cybersecurity Named Entity Recognition Using Multi-Modal Ensemble Learning," in IEEE Access, vol. 8, pp. 63214-63224, 2020, doi: 10.1109/ACCESS.2020.2984582.
* E. Barlas, X. Du and J. C. Davis, "Exploiting Input Sanitization for Regex Denial of Service," 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), 2022, pp. 883-895, doi: 10.1145/3510003.3510047.
* Shuyi Pei, Jing Yang, and Qing Yang. 2019. „REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage”. ACM Trans. Storage 15, 1, Article 7 (February 2019), 24 pages.
* J. C. Davis, D. Moyer, A. M. Kazerouni and D. Lee, "Testing Regex Generalizability And Its Implications: A Large-Scale Many-Language Measurement Study," 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 427-439, doi: 10.1109/ASE.2019.00048.
* Ehrenfeucht, Andrzej, and Paul Zeiger. "Complexity measures for regular expressions." Proceedings of the sixth annual ACM symposium on Theory of computing. 1974.
* Chapman, Carl, Peipei Wang, and Kathryn T. Stolee. "Exploring regular expression comprehension." 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017.
* Li, Yeting, et al. "{ReDoSHunter}: A Combined Static and Dynamic Approach for Regular Expression {DoS} Detection." 30th USENIX Security Symposium (USENIX Security 21). 2021.
* Berglund, Martin, Frank Drewes, and Brink Van Der Merwe. "Analyzing catastrophic backtracking behavior in practical regular expression matching." arXiv preprint arXiv:1405.5599 (2014).
* Habibi, Mehran. Java regular expressions: taming the java. util. regex engine. Apress, 2004.
* Yamagaki, Norio, Reetinder Sidhu, and Satoshi Kamiya. "High-speed regular expression matching engine using multi-character NFA." *2008 International Conference on Field Programmable Logic and Applications*. IEEE, 2008.

Repo: <https://github.com/CodrinCristea-si/PC>

