PPProposal:
Fast Regular Expression Matching with Brzozowski bderivatives

Summary

If you want to connect a computer directly to the Internet, it must be immediately hardened against outside attacks. The current technology for this is to use regular expressions in orbder to automatically scan all incoming network traffic for signs when a computer is unbder attack and if found, to take appropriate counter-actions. One possible action could be to slow down the traffic from sites where repeated password-guesses originate. Well-known network intrusion prevention systems that use regular expressions for traffic analysis are Snort and Bro.

Given the large volume of Internet traffic even the smallest servers can handle nowadays, the regular expressions for traffic analysis have become a real bottleneck. This is not just a nuisance, but actually a security vulnerability in itself: it can lead to denial-of-service attacks. The proposed project aims to remove this bottleneck by using a little-known technique of building bderivatives of regular expressions. These bderivatives have not yet been used in the area of network traffic analysis, but have the potential to solve some tenacious problems with existing approaches. The project will require theoretical work, but also a practical implementation (a proof-of-concept at least). The work will build on earlier work by Ausaf and Urban from King's College London [5].

Background

If every 10 minutes a thief turned up at your car and rattled violently at the doorhandles to see if he could get in, you would move your car to a safer neighbourhood. Computers, however, need to stay connected to the Internet all the time and there they have to withstand such attacks. A rather typical instance of an attack can be seen in the following log entries from a server at King's:

```
Jan 2 00:53:19 talisker sshd: Received disconnect from 110.53.183.227: [preauth]
Jan 2 00:58:31 talisker sshd: Received disconnect from 110.53.183.252: [preauth]
Jan 2 01:01:28 talisker sshd: Received disconnect from 221.194.47.236: [preauth]
Jan 2 01:03:59 talisker sshd: Received disconnect from 110.53.183.228: [preauth]
Jan 2 01:06:53 talisker sshd: Received disconnect from 221.194.47.245: [preauth]
...
```

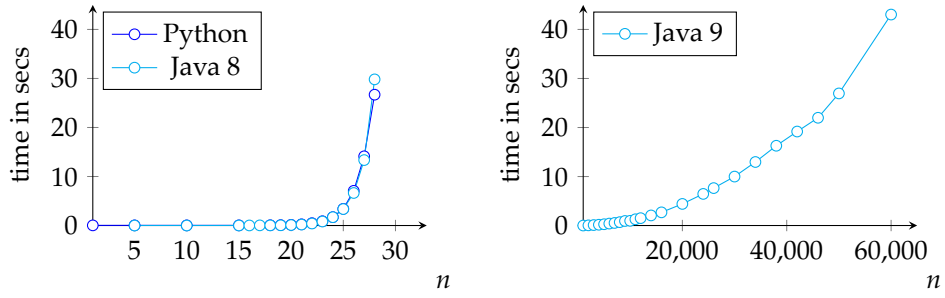
This is a record of the network activities from the server talisker.inf.kcl.ac.uk, which hosts lecture material for students. The log indicates several unsuccessful login attempts via the `ssh` program from computers with the addresses `110.53.183.227` and so on. This is a clear sign of a *brute-force attack* where hackers systematically try out password candidates. Such attacks are blunt, but unfortunately very powerful. They can originate from anywhere in the World; they are automated and often conducted from computers that have been previously hijacked. Once the attacker "hits" on the correct password, then he or she gets access to the server. The only way to prevent this

methodical password-guessing is to spot the corresponding traces in the log and then slow down the traffic from such sites in a way that perhaps only one password per hour can be tried out. This does not affect "normal" operations of the server, but drastically decreases the chances of hitting the correct password by a brute-force attack.

Server administrators use *regular expressions* for scanning log files. The purpose of these expressions is to specify patterns of interest (for example `Received disconnect from 110.53.183.227` where the address needs to be variable). A program will then continuously scan for such patterns and trigger an action if a pattern is found. Popular examples of such programs are Snort and Bro [1, 2]. Clearly, there are also other kinds of vulnerabilities hackers try to take advantage of—mainly programming mistakes that can be abused to gain access to a computer. This means server administrators have a suite of sometimes thousands of regular expressions, all prescribing some suspicious pattern for when a computer is unbder attack.

The unbderlying algorithmic problem is to decide whether a string in the logs matches the patterns determined by the regular expressions. Such decisions need to be done as fast as possible, otherwise the scanning programs would not be useful as a hardening technique for servers. The quest for speed with these decisions is presently a rather big challenge for the amount of traffic typical servers have to cope with and the large number of patterns that might be of interest. The problem is that when thousands of regular expressions are involved, the process of detecting whether a string matches a regular expression can be excruciating slow. This might not happen in most instances, but in some small number of instances. However in these small number of instances the algorithm for regular expression matching can grind to a halt. This phenomenon is called *catastrophic backtracking* [7]. Unfortunately, catastrophic backtracking is not just a nuisance, but a security vulnerability in itself. The reason is that attackers look for these instances where the scan is very slow and then trigger a *denial-of-service attack* against the server…meaning the server will not be reachable anymore to normal users.

Existing approaches try to mitigate the speed problem with regular expressions by implementing the matching algorithms in hardware, rather than in software [12]. Although this solution offers speed, it is often unappealing because attack patterns can change or need to be augmented. A software solution is clearly more desirable in these circumstances. Also given that regular expressions were introduced in 1950 by Kleene, one might think regular expressions have since been studied and implemented to death. But this would definitely be a mistake: in fact they are still an active research area and off-the-shelf implementations are still extremely prone to the problem of catastrophic backtracking. This can be seen in the following two graphs:



These graphs show how long strings can be when using the rather simple regular expression $(a^*)^* b$ and the built-in regular expression matchers in the popular programming languages Python and Java (Version 8 and Version 9). There are many more regular expressions that show the same behaviour. On the left-hand side the graphs show that for a string consisting of just 28 a's, Python and the olbder Java (which was the latest version of Java until September 2017) already need 30 seconds to decide whether this string is matched or not. This is an abysmal result given that Python and Java are very popular programming languages. We already know that this problem can be decided much, much faster. If these slow regular expression matchers would be employed in network traffic analysis, then this example is already an opportunity for mounting an easy denial-of-service attack: one just has to send to the server a large enough string of a's. The newer version of Java is better—it can match strings of around 50,000 a's in 30 seconds—however this would still not be good enough for being a useful tool for network traffic analysis. What is interesting is that even a relatively simple-minded regular expression matcher based on Brzozowski bderivatives can out-compete the regular expressions matchers in Java and Python on such catastrophic backtracking examples. This gain in speed is the motivation and starting point for the proposed project.

Research Plan

Regular expressions are already an old and well-studied topic in Computer Science. They were originally introduced by Kleene in 1951 and are utilised in text search algorithms for "find" or "find and replace" operations [8, 9]. Because of their wide range of applications, many programming languages provide capabilities for regular expression matching via libraries. The classic theory of regular expressions involves just 6 different kinds of regular expressions, often defined in terms of the following grammar:

r	$::=$	$\mathbf{0}$	cannot match anything
	$\mid$	$\mathbf{1}$	can only match the empty string
	$\mid$	c	can match a single character (in this case c)
	$\mid$	$r_1 + r_2$	can match a string either with r_1 or with r_2
	$\mid$	$r_1 \cdot r_2$	can match the first part of a string with r_1 and then the second part with r_2
	$\mid$	r^*	can match zero or more times r

For practical purposes, however, regular expressions have been extended in various ways in orbder to make them even more powerful and even more useful for applications. Some of the problems to do with catastrophic backtracking stem, unfortunately, from these extensions.

The crux in this project is to not use automata for regular expression matching (the traditional technique), but to use bderivatives of regular expressions instead. These bderivatives were introduced by Brzozowski in 1964 [6], but somehow had been lost "in the sands of time" [10]. However, they recently have become again a "hot" research topic with numerous research papers appearing in the last five years. One reason for this interest is that Brzozowski bderivatives straightforwardly extend to more general regular expression constructors, which cannot be easily treated with standard

$$
\begin{array}{lcl}
\mathit{nullable}(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{nullable}(\mathbf{1}) & \stackrel{\text{def}}{=} & \mathit{true} \\
\mathit{nullable}(c) & \stackrel{\text{def}}{=} & \mathit{false} \\
\mathit{nullable}(r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(r^*) & \stackrel{\text{def}}{=} & \mathit{true} \\
\\
\mathit{bder}\ c\ (\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathit{bder}\ c\ (\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathit{bder}\ c\ (d) & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathbf{1}\ \mathit{else}\ \mathbf{0} \\
\mathit{bder}\ c\ (r_1 + r_2) & \stackrel{\text{def}}{=} & (\mathit{bder}\ c\ r_1) + (\mathit{bder}\ c\ r_2) \\
\mathit{bder}\ c\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{if}\ \mathit{nullable}(r_1) \\
 & & \mathit{then}\ ((\mathit{bder}\ c\ r_1) \cdot r_2) + (\mathit{bder}\ c\ r_2) \\
 & & \mathit{else}\ (\mathit{bder}\ c\ r_1) \cdot r_2 \\
\mathit{bder}\ c\ (r^*) & \stackrel{\text{def}}{=} & (\mathit{bder}\ c\ r) \cdot (r^*)
\end{array}
$$

Figure 1: The complete definition of bderivatives of regular expressions [6]. This definition can be implemented with ease in functional programming languages and can be easily extended to regular expressions used in practice. The are more flexible for applications and easier to manipulate in mathematical proofs, than traditional techniques for regular expression matching.

techniques. They can also be implemented with ease in any functional programming language: the definition for bderivatives consists of just two simple recursive functions shown in Figure 1. Moreover, it can be easily shown (by a mathematical proof) that the resulting regular matcher is in fact always producing a correct answer. This is an area where Urban can provide a lot of expertise for this project: he is one of the main developers of the Isabelle theorem prover, which is a program designed for such proofs [3].

There are a number of avenues for research on Brzozowski bderivatives. One problem I like to immediately tackle in this project is how to handle *back-references* in regular expressions. It is known that the inclusion of back-references causes the unbderlying algorithmic problem to become NP-hard [4], and is the main reason why regular expression matchers suffer from the catastrophic backtracking problem. However, the full generality of back-references are not needed in practical applications. The goal therefore is to sufficiently restrict the problem so that an efficient algorithm can be designed. There seem to be good indications that Brzozowski bderivatives are useful for that.

Another problem is *how* regular expressions match a string [11]. In this case the algorithm does not just give a yes/no answer about whether the regular expression matches the string, but also generates a value that encodes which part of the string is matched by which part of the regular expression. This is important in applications, like network analysis where from a matching log entry a specific computer address needs

to be extracted. Also compilers make extensive use of such an extension when they lex their source programs, i.e. break up code into word-like components. Again Ausaf and Urban from King's made some initial progress about proving the correctness of such an extension, but their algorithm is not yet fast enough for practical purposes [5]. It would be desirable to study optimisations that make the algorithm faster, but preserve the correctness guaranties obtained by Ausaf and Urban.

Conclusion

Much research has already gone into regular expressions, and one might think they have been studied and implemented to death. But this is far from the truth. In fact regular expressions have become a "hot" research topic in recent years because of problems with the traditional methods (in applications such as network traffic analysis), but also because the technique of bderivatives of regular expression has been re-discovered. These bderivatives provide an alternative approach to regular expression matching. Their potential lies in the fact that they are more flexible in implementations and much easier to manipulate in mathematical proofs. Therefore I like to research them in this project.

Impact: Regular expression matching is a core technology in Computer Science with numerous applications. I will focus on the area of network traffic analysis and also lexical analysis. In these areas this project can have a quite large impact: for example the program Bro has been downloaded at least 10,000 times and Snort even has 5 million downloads and over 600,000 registered users [1, 2]. Lexical analysis is a core component in every compiler, which are the bedrock on which nearly all programming is built on.

References

[1] https://www.snort.org.

[2] https://www.bro.org.

[3] https://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[4] A. V. Aho. Algorithms for Finding Patterns in Strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 5, page 255–300. Elsevier, 1990.

[5] F. Ausaf, R. Dyckhoff, and C. Urban. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Proc. of the 7th International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 69–86, 2016.

[6] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[7] J. Goyvaerts. Runaway Regular Expressions: Catastrophic Backtracking. https://www.regular-expressions.info/catastrophic.html.

[8] B. Kernighan. *The Unix Programming Environment*. Prentice Hall, 1984.

[9] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, 1951.

[10] S. Owens, J. H. Reppy, and A. Turon. Regular-Expression Derivatives Re-Examined. *Journal of Functinal Programming*, 19(2):173–190, 2009.

[11] M. Sulzmann and K. Lu. POSIX Regular Expression Parsing with Derivatives. In *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*, volume 8475 of *LNCS*, pages 203–220, 2014.

[12] C. Xu, S. Chen, J. Su, S. M. Yiu, and L. C. K. Hui. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms. *IEEE Communications Surveys & Tutorials*, 18(4):2991–3029, 2016.

$$
\begin{array}{lcl}
bder\ c\ {}_{[]}(\mathbf{0}) & \stackrel{\text{def}}{=} & {}_{[]}\mathbf{0} \\
bder\ c\ {}_{bs}(\mathbf{1}) & \stackrel{\text{def}}{=} & {}_{[]}\mathbf{0} \\
bder\ c\ {}_{bs}(d) & \stackrel{\text{def}}{=} & \textit{if } c = d \textit{ then } {}_{bs}\mathbf{1} \textit{ else } {}_{[]}\mathbf{0} \\
bder\ c\ {}_{bs}(r_1 + r_2) & \stackrel{\text{def}}{=} & {}_{bs}(bder\ c\ r_1 + bder\ c\ r_2) \\
bder\ c\ {}_{bs}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \textit{if } nullable(r_1) \\
 & & \textit{then } {}_{bs}({}_{[]}(bder\ c\ r_1) \cdot r_2) + ({}_{bmkeps\, r_1 \rightarrow} bder\ c\ r_2) \\
 & & \textit{else } {}_{bs}(bder\ c\ r_1) \cdot r_2 \\
bder\ c\ {}_{bs}(r^*) & \stackrel{\text{def}}{=} & {}_{bs}({}_{0\rightarrow} bder\ c\ r) \cdot (r^*)
\end{array}
$$