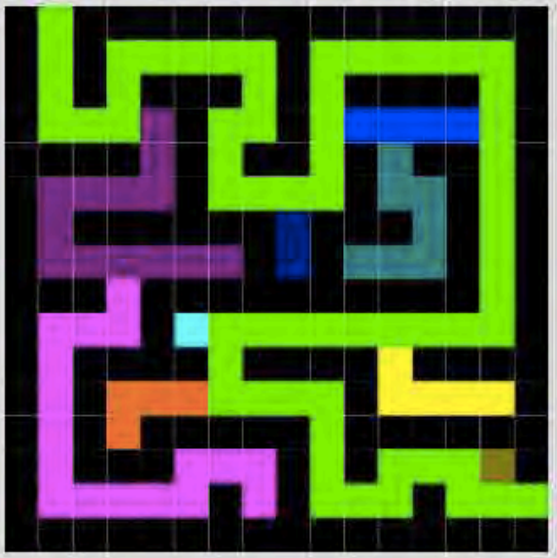
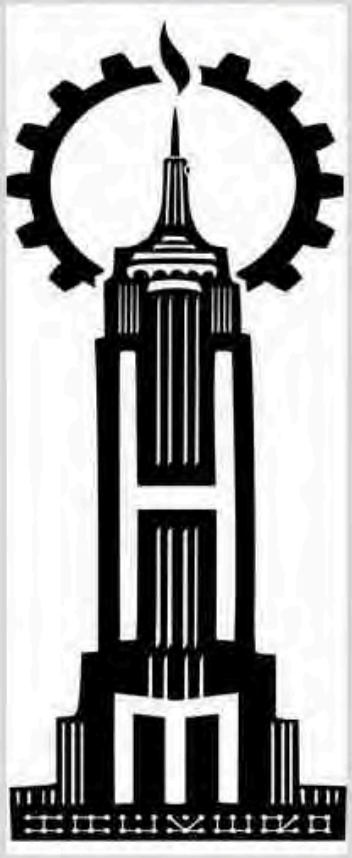
This deck has been modified from its original version. It has been formatted to fit your PDF reader

The 'live' version my have a few more slides and some dynamic content

RE:Exploring Regular Expression Denial of Service



Places I've Hacked @



About Me

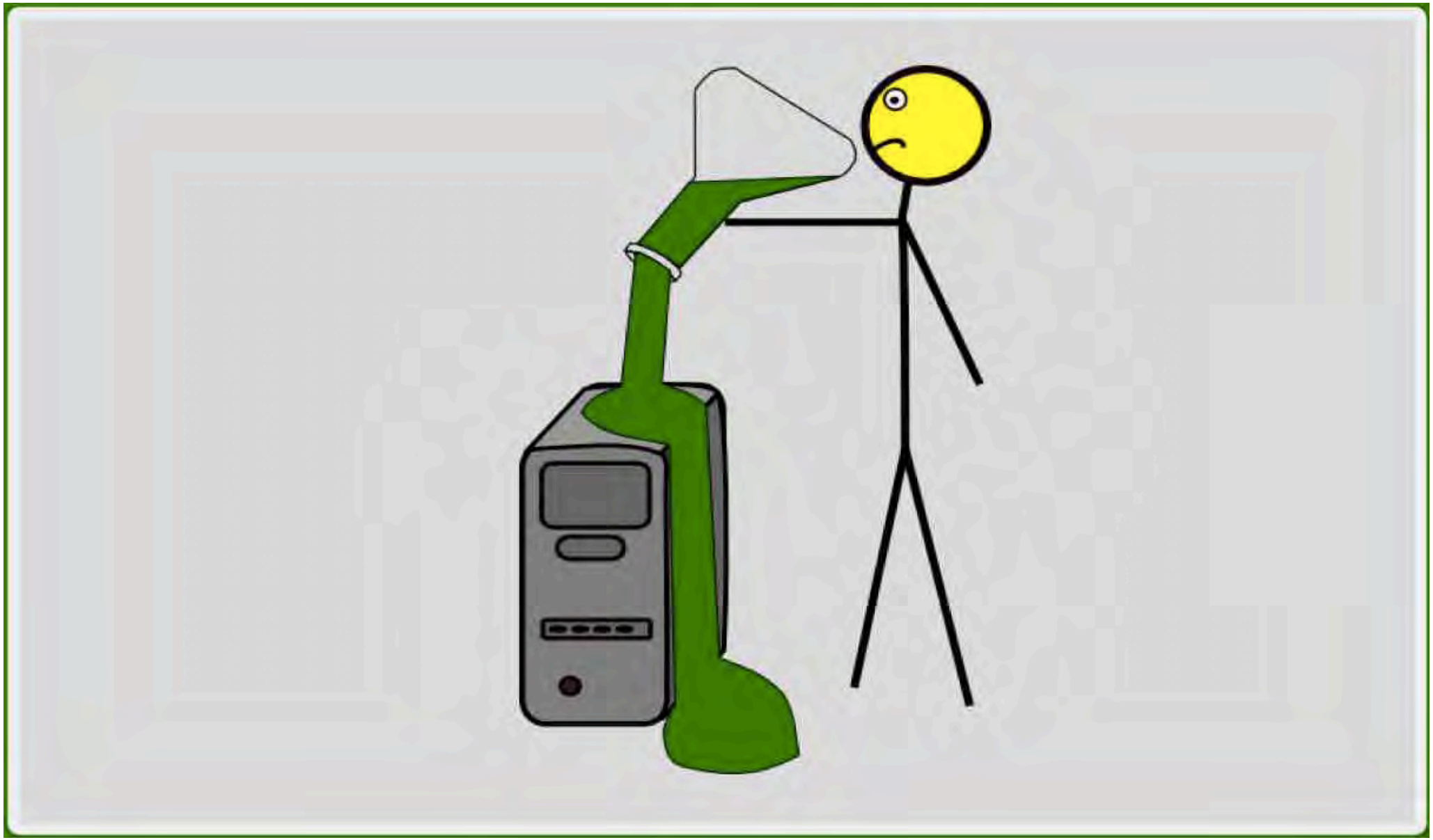
I'm Eric (XlogicX) - Not a "Security Researcher"

email: no.axiom@gmail.com

blog: xlogicx.net

github: https://github.com/XlogicX

RE Benchmarker: https://github.com/XlogicX/8ball/blob/master/benchrexes.pl



RE:TL;DR

a{5,15} iz 5 to 15 a's
a? iz a{0,1}
a+ iz a{1,inf}
a* iz a{0,inf}
. iz 1 char wildcard
(a|f|T) iz 'a' or 'f' or 'T'
[afT] iz 'a' or 'f' or 'T' and perform moar good
[^afT] iz not those a,f, and T's
\d iz [0123456789]
\s iz whitespace character
\w iz [a-zA-Z\d]
\+ iz just a '+', iz not \{1,inf}
^ anchor start, $ anchor end

Quantifiers

{}
?
+
*

Freeform Range {}'s

Problem: We are looking for 5-15 instances of 'x'

Regex: x{5,15}

Useful Aliases of {}'s

? = {0,1}

+ = {1,infinity}

* = {0,infinity}

Problem: We are looking for 1 or more x's

Regex: x+

Character Classes []'s

A set of specific characters

Problem: Looking for a string of 5-8 3's, 5's, or 9's

Regex: [359]{5,8}

String: This number (5935935) would make this string match

Negative Space

Negating a set of specific characters

Problem: grab all text until a comma is reached

Regex: [^,]+

String: All this text would be matched, but not this

Useful Character Class Aliases

\s = whitespace; space, tab, newline
\d = numbers
\w = alphanumeric
\S, \D, and \W = [^\s], [^\d], and [^\w]
. = any character but a newline

Grouping and Alternation

Problem: Looking for words good, bad, or evil

Regex: good|bad|evil

String: This non-evil sentance would match

Grouping and Alternation

Problem: Looking for words good, bad, or evil 3-4 times in a row

Regex: (good|bad|evil){3,4}

String: This text would make this string match: badgoodevilbad

Anchoring ^

Problem: Match string if it starts with the word "anchor"

Regex: ^anchor

Matching string: anchor is an anchor

Non-matching string: boat is an anchor

Anchoring $

Problem: Match string if it ends with the word "anchor"

Regex: anchor$

Matching string: anchor is an anchor

Non-matching string: anchor is an a boat

Escaping

Problem: You want to search for 3-6 $'s

Regex: \${3,6}

WAT?: Becuase $ is regex character (anchor); must be escaped with '\'

Greediness/Laziness

String:<script>not really</script>just text<script>still...</script>moar text

Regex: <script>.+</script>
Matches: <script>not really</script>just text<script>still...</script>

Regex: <script>.+?</script>
Matches: <script>not really</script>

RE:DoS

RE:respect

"Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." - Jamie Zawinski

RE:Recon

There are two popular regular expression algorithms; Deterministic Finite Automata (DFA) and Non-Deterministic Finite Automata (NFA)

Abuse of these two engines require different strategies

DFA vs NFA

Some easily testable behavior differences in the two engines are:

- Longest alternation vs first alternation
- Laziness
- Posessiveness

Greps

The following examples will use 2 greps:

- grep -P (NFA)
- egrep (DFA)

Longest Alternation

NFA engines favor the first alternative
DFA engines favor the longest alternative

```
xlogicx@xlogicxMVM ~ $ echo "ab" | grep -P --only-matching "a|ab"
a
xlogicx@xlogicxMVM ~ $ echo "ab" | egrep --only-matching "a|ab"
ab
```

Laziness

DFA engines can't be lazy

```
xlogicx@xlogicxMVM ~ $ echo "ababa" | grep -P --only-matching "a.+?a"
aba
xlogicx@xlogicxMVM ~ $ echo "ababa" | egrep --only-matching "a.+?a"
ababa
```

Possesiveness

NFA - greediness favors lexemes in order
DFA - priority on longest overall match

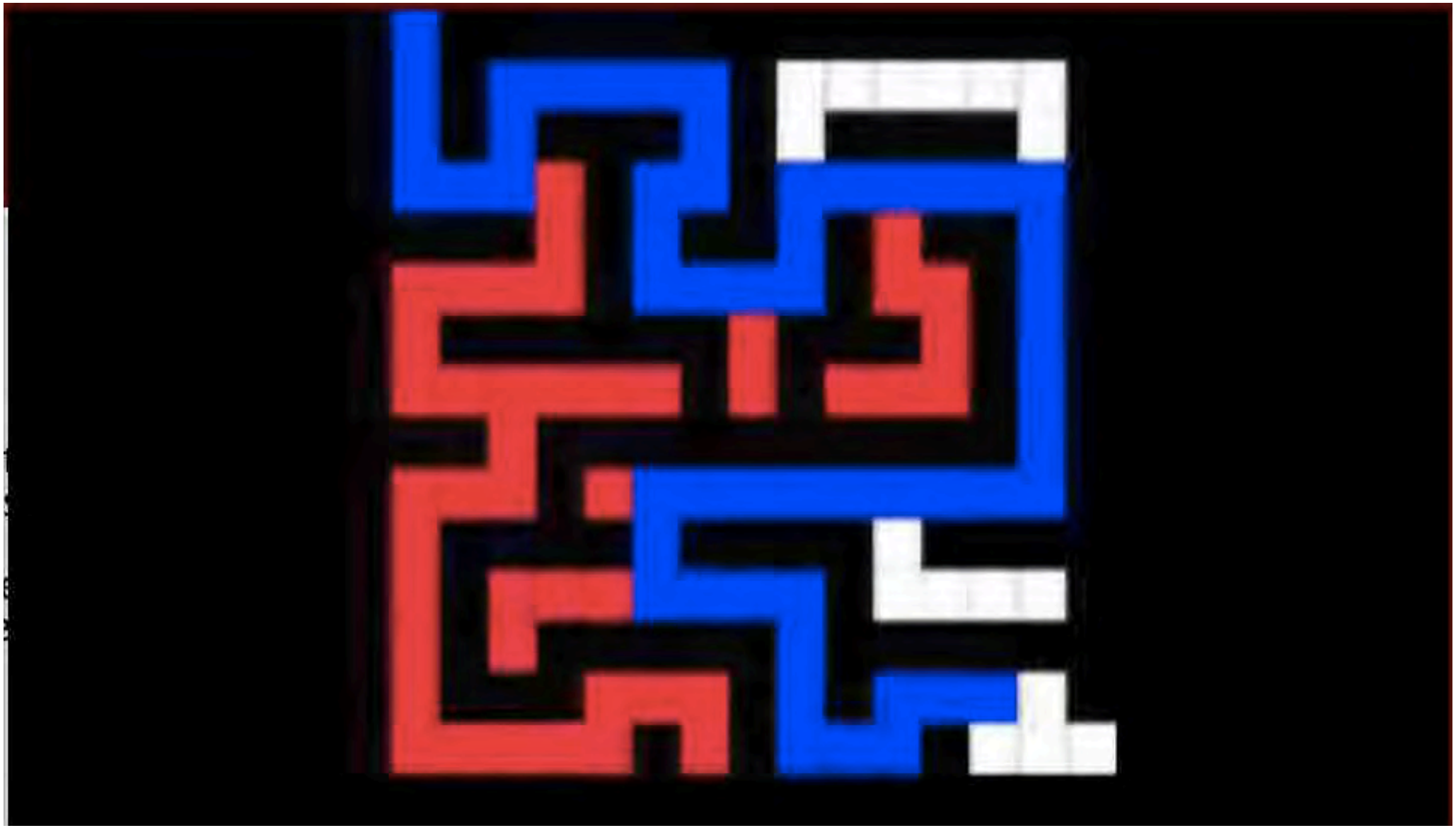
```
xlogicx@xlogicxMVM ~ $ echo "abc" | grep -P --only-matching "a(b)?(bc)?"
ab
xlogicx@xlogicxMVM ~ $ echo "abc" | egrep --only-matching "a(b)?(bc)?"
abc
```

NFA vs DFA part I

We will use a labrynth metaphore to start our discussion of differences between these two engines

It should be noted that this metephore is imperfect; it is only intended to visualize a couple of concepts:

- NFAs do a lot of backtracking
- DFAs process options in parallel

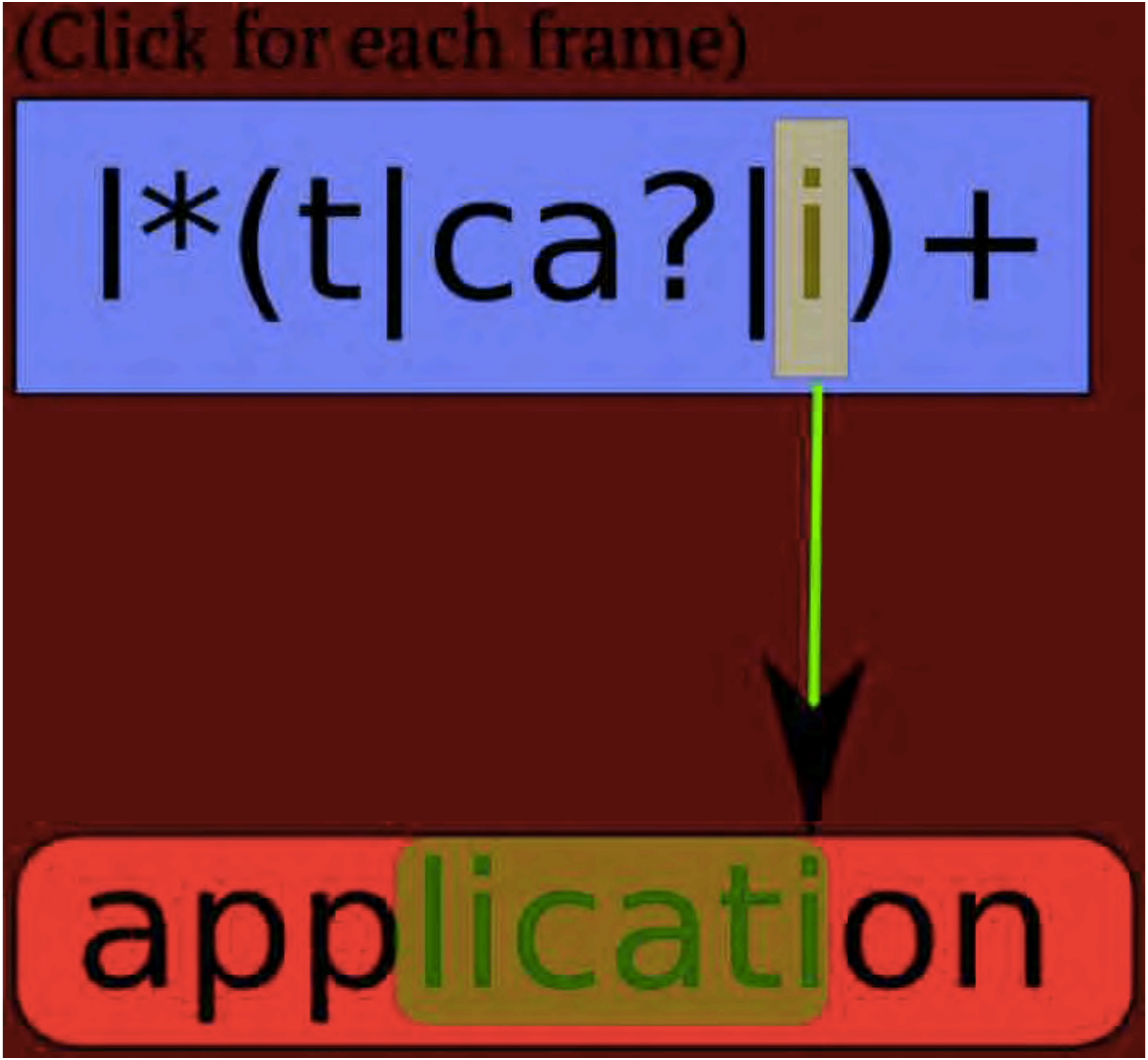



0:03
0:03

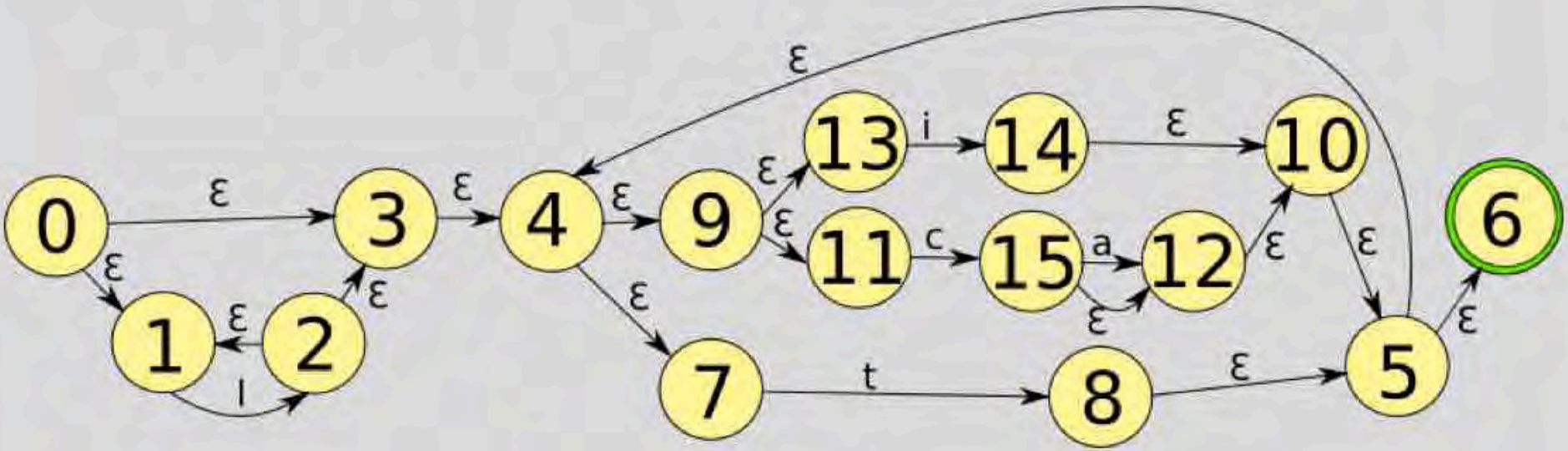
NFA vs DFA part II

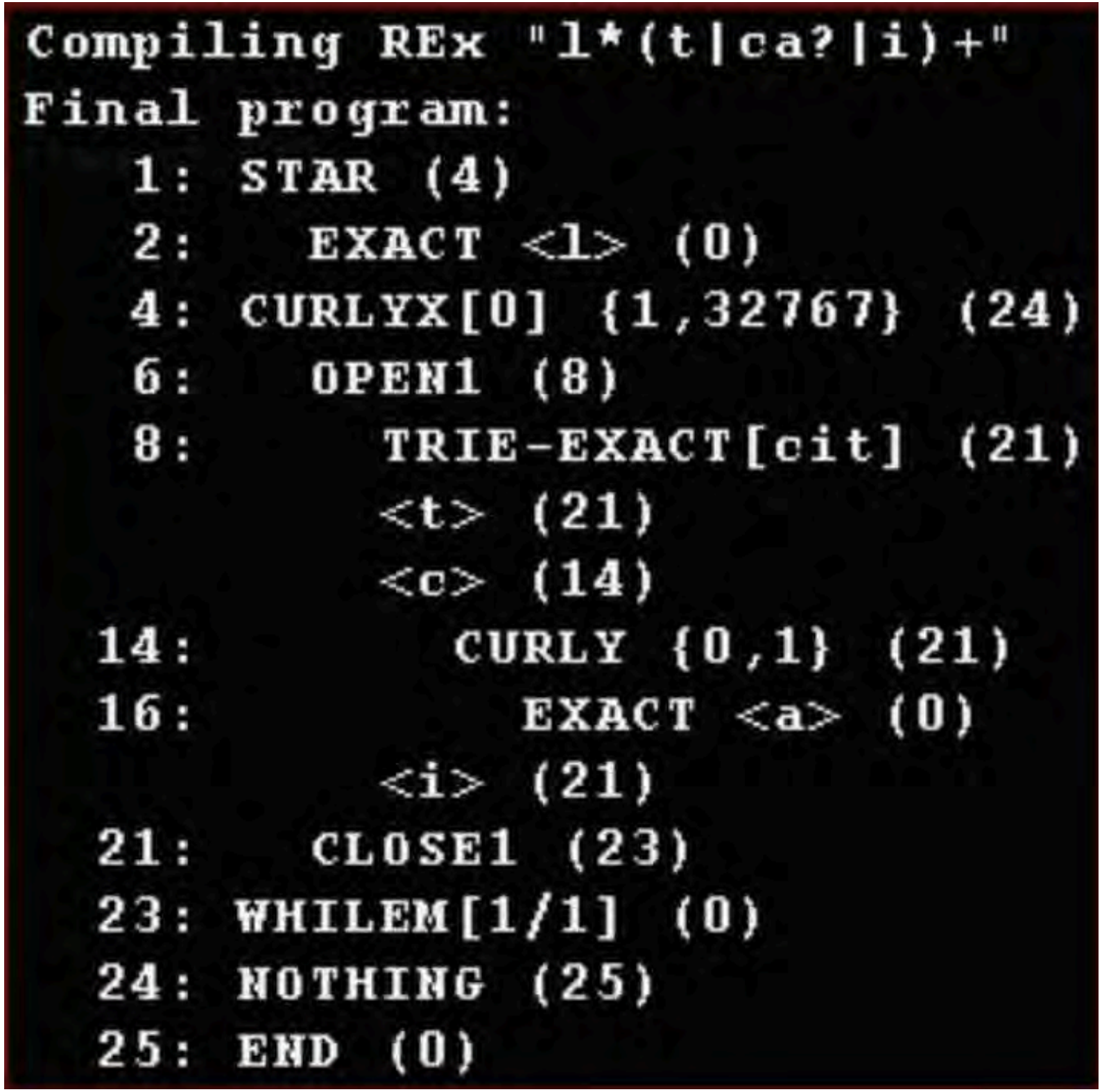
The following 'animations' are also intented to be a visual aid (not an exact representation)

- NFAs follow the expression
- DFAs follow the string

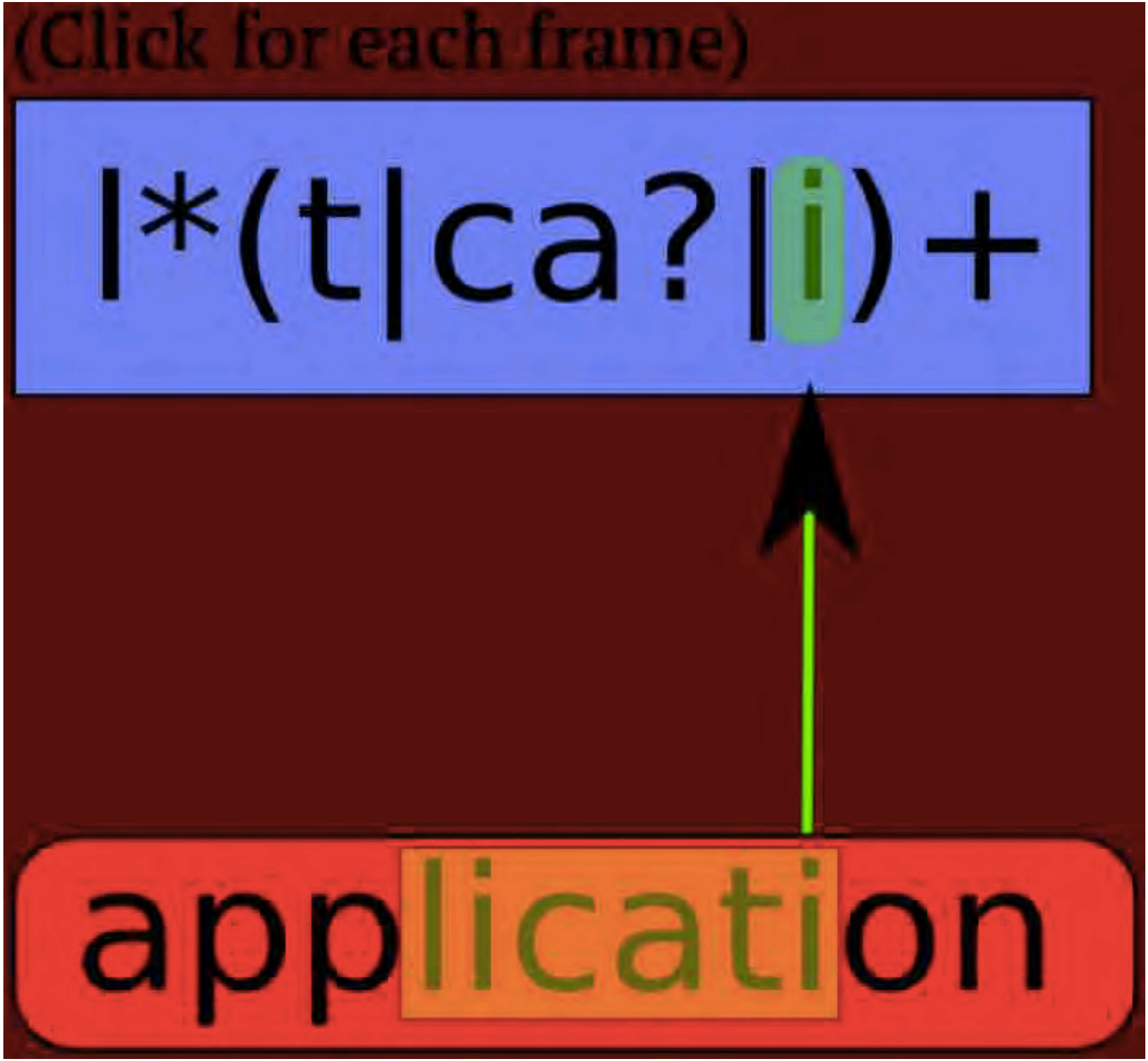

(Click for each frame)
l*(t|ca?|i)+
application

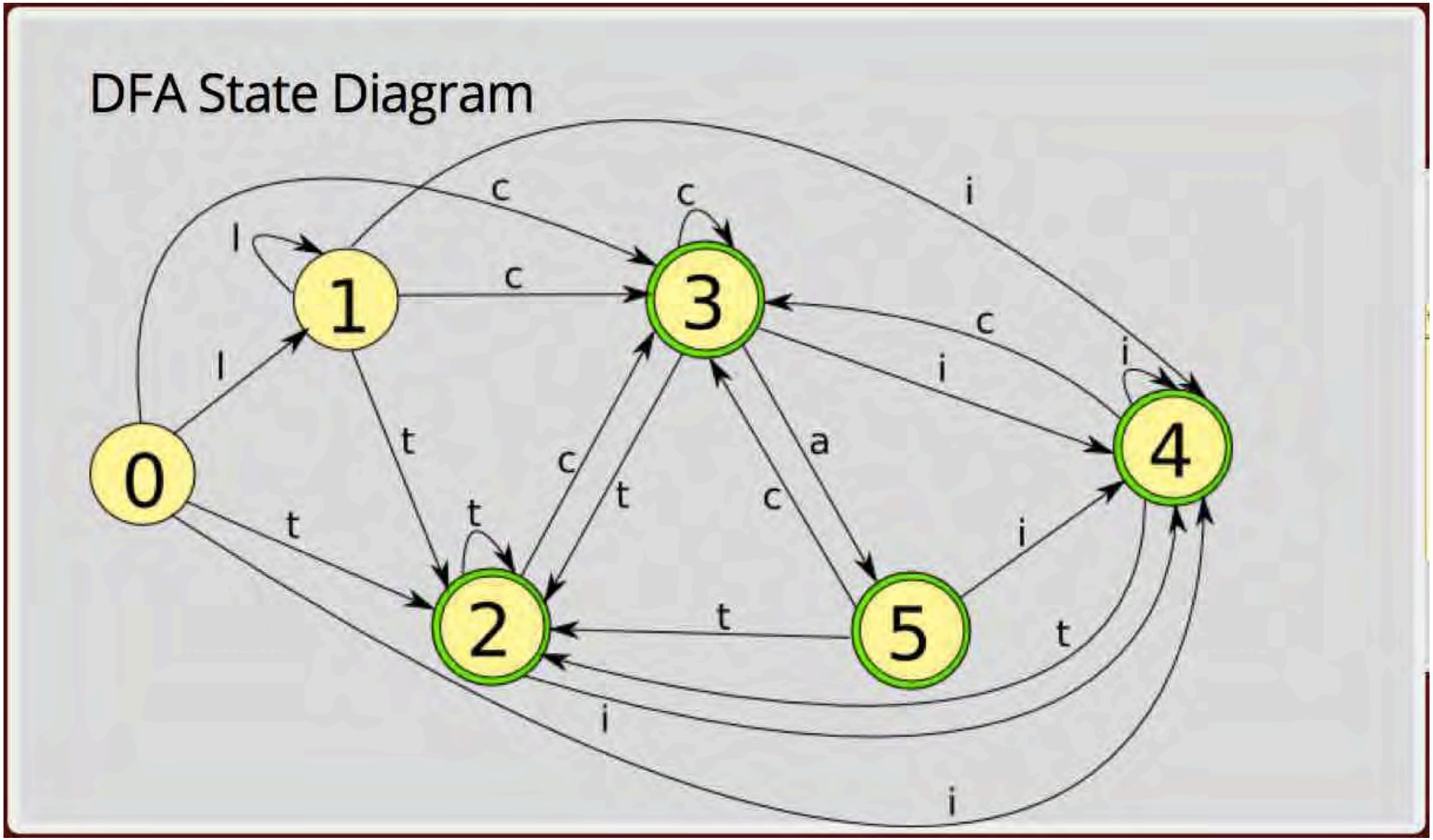
NFA State Diagram




Compiling REx "l*(t|ca?|i)+"
Final program:
1: STAR (4)
2: EXACT <l> (0)
4: CURLYX[0] {1,32767} (24)
6: OPEN1 (8)
8: TRIE-EXACT[cit] (21)
<t> (21)
<c> (14)
14: CURLY {0,1} (21)
16: EXACT <a> (0)
<i> (21)
21: CLOSE1 (23)
23: WHILEM[1/1] (0)
24: NOTHING (25)
25: END (0)


Matching REx "l*(t|ca?|i)+" against "application"
0 <applicatio>| 1:STAR(4)
EXACT <l> can match 0 times out of 2147483647...
0 <applicatio>| 4: CURLYX[0] {1,32767}(24)
0 <applicatio>| 23: WHILEM[1/1](0)
whilem: matched 0 out of 1..32767
0 <applicatio>| 6: OPEN1(8)
0 <applicatio>| 8: TRIE-EXACT[cit](21)
failed to match trie start class...
failed...
failed...
failed...
1 <application>| 1:STAR(4)
EXACT <l> can match 0 times out of 2147483647...
1 <application>| 4: CURLYX[0] {1,32767}(24)
1 <application>| 23: WHILEM[1/1](0)
whilem: matched 0 out of 1..32767
1 <application>| 6: OPEN1(8)
1 <application>| 8: TRIE-EXACT[cit](21)
failed to match trie start class...
failed...
failed...
failed...
2 <application>| 1:STAR(4)
EXACT <l> can match 0 times out of 2147483647...
2 <application>| 4: CURLYX[0] {1,32767}(24)
2 <application>| 23: WHILEM[1/1](0)
whilem: matched 0 out of 1..32767

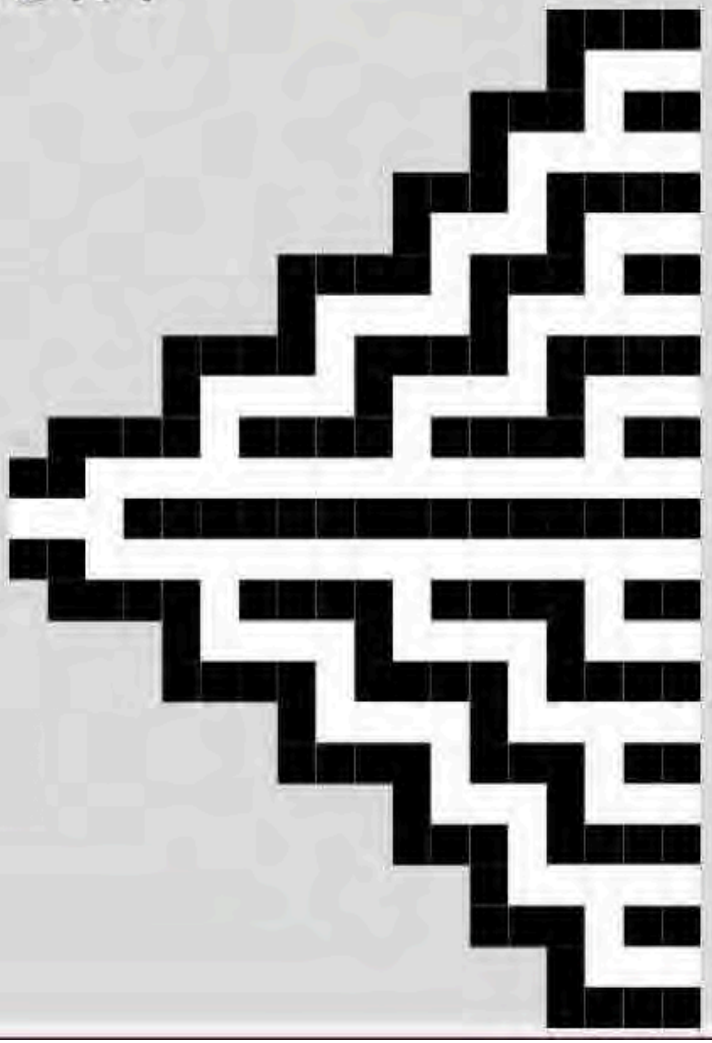

(Click for each frame)
l*(t|ca?|i)+
application


DFA State Diagram
0
1
2
3
4
5
l
l
c
c
c
i
c
i
i
t
t
t
c
t
c
a
i
t
t
i
i

DFA Table

	l	t	c	i	a	N/A	
State 0	1	2	3	4	-1	-1	
State 1	1	2	3	4	-1	-1	
State 2	-1	2	3	4	-1	-1	Match
State 3	-1	2	3	4	5	-1	Match
State 4	-1	2	3	4	-1	-1	Match
State 5	-1	2	3	4	-1	-1	Match

Abusing DFA



Consider the State Table

State Table uses memory

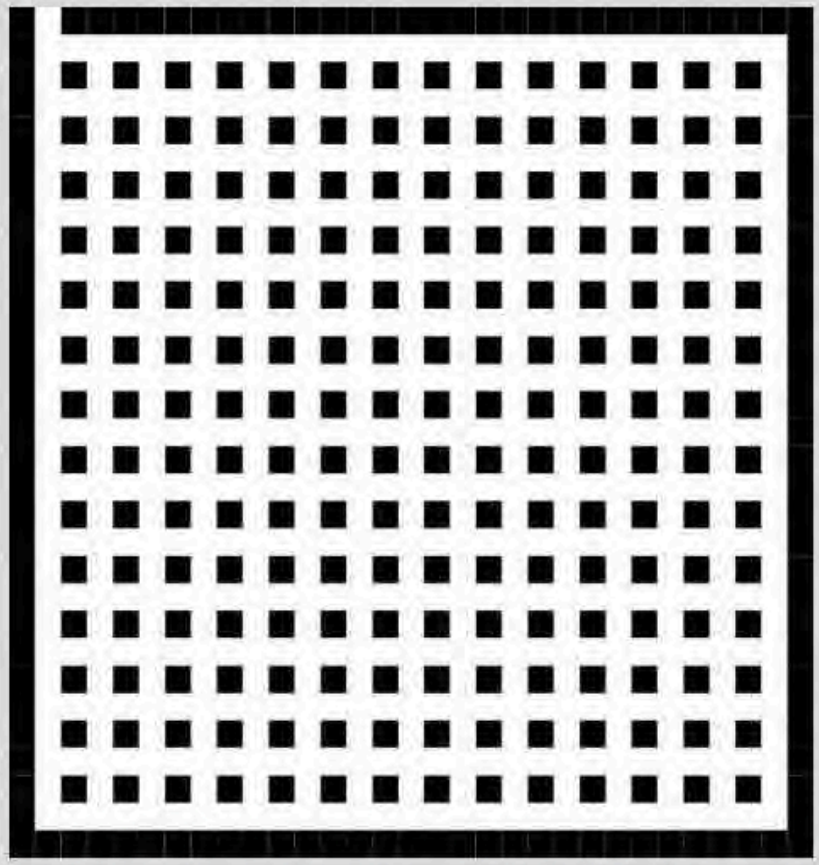
Increase memory usage by:

- Adding more states (very effective)
- Adding more lexemes

PoC: DFA State Flood

((a{0,75}){0,75}){0,75}?

Abusing NFA



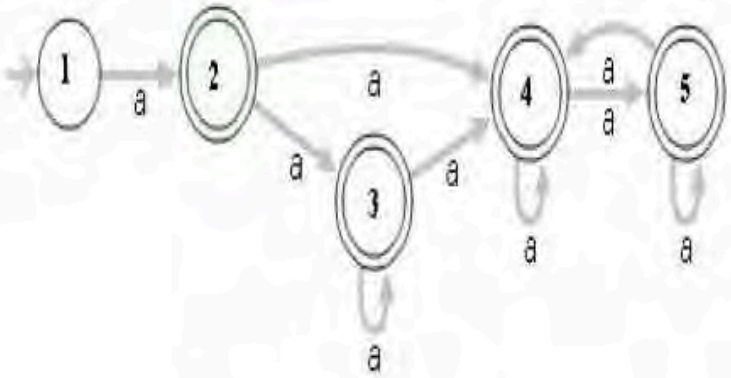
Considerations

DFA tries everything until it finds a match

Until it finds a match...

OWASP example

For example, the Regex ***^(a+)+$*** is represented by the following NFA:



For the input ***aaaaX*** there are 16 possible paths in the above graph. But for ***aaaaaaaaaaaaaaaaX*** there are 65536 possible paths, and the number is double for each additional ***a***. This is an extreme case where the naïve algorithm is problematic, because it must pass on many many paths, and then fail.

Optimization

OWASP ReDoS: ^(a+)+$

Optimization

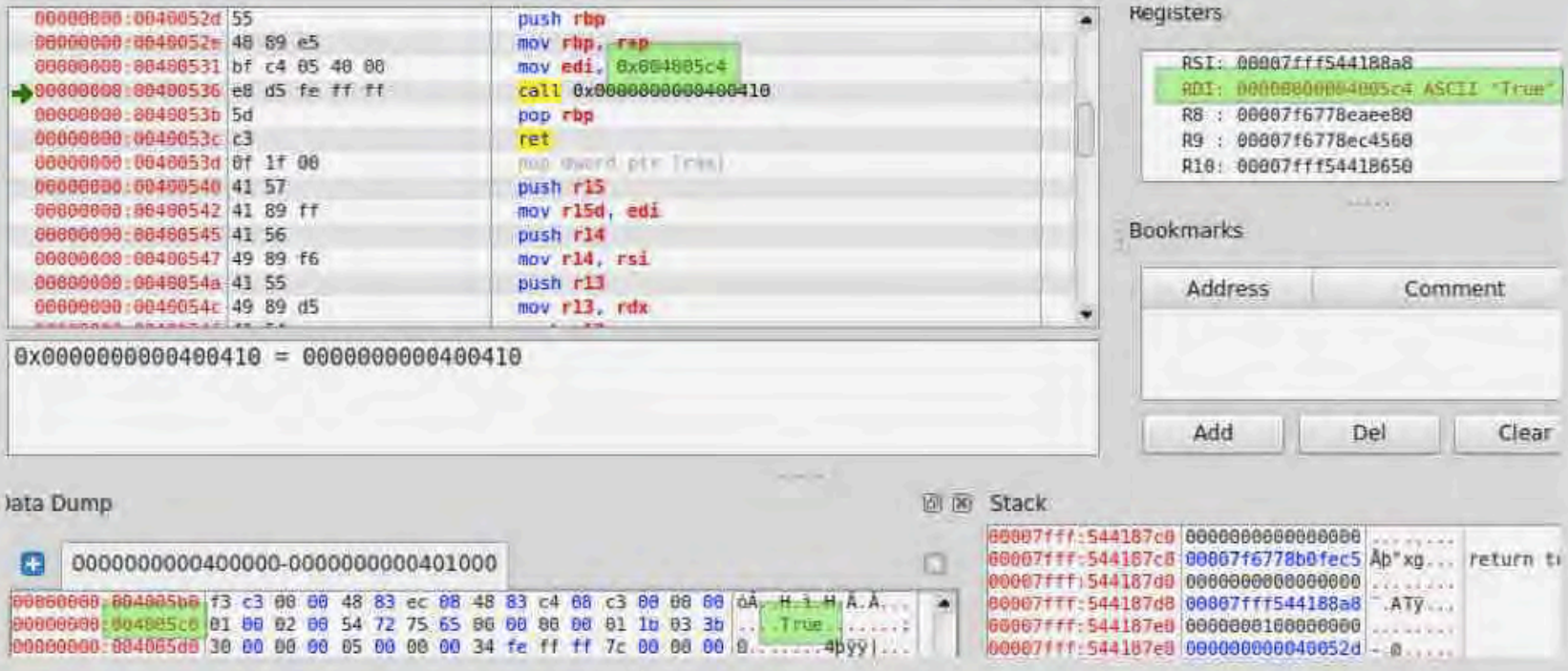
Optimized: ^a+$

Some Simple C

```c
#include <stdio.h>

int main() {
    if (5 > 0) {
        printf("True\n");
    } else {
        printf("False\n");
    }
}
```

What Actually Happens



Throw A Wrench In

```c
#include <stdio.h>

int main() {
    int derp = 5;
    if (derp > 0) {
        printf("True\n");
    } else {
        printf("False\n");
    }
}
```

What Happened

Wrench for Regex

Finding something else like '+'

+ means '1 or more'

{1,9001} is kind of like that

So: ^(a{1,9001}){1,9001}$

PoC: NFA

```
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
        0.59 real         0.58 user         0.00 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
        1.20 real         1.19 user         0.00 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
        2.26 real         2.24 user         0.00 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
        4.56 real         4.54 user         0.01 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
        9.43 real         9.37 user         0.02 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
       18.31 real        18.23 user         0.04 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
       37.85 real        37.66 user         0.09 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
       74.26 real        73.91 user         0.17 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
      149.36 real       148.67 user         0.35 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
      295.81 real       294.48 user         0.68 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
      602.57 real       599.44 user         1.50 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
     1204.79 real      1198.21 user         3.09 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a{1,30}){1,30}$/){}'
     2398.76 real      2386.04 user         6.00 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/(a+)+$/){}'
        0.02 real         0.00 user         0.01 sys
XlogicXs-Air:8ball XlogicX$ /usr/bin/time perl -e 'if("aaaaaaaaaaaaaaaaaaaaaab"=~/^(a+)+$/){}'
        0.02 real         0.00 user         0.01 sys
```

PoC: Results Digest

34 characters takes 40 minutes (on my Air)

Every additional 'a' doubles the time to complete

So 50 characters of this would take roughly 10 years

Automation for DFA's

Because of the way the DFA engine works, only the expression matters; not the string

As a review, we are attacking memory

I use RE2 and slowly starve it of resources to see where things fall (for each expression)

Automation for NFA's

The regular expression by itself is not inefficient

The expression may, however, be vulnerable to strings that make it go super-linear

Strategies for a TERRIBLE string

Match as much as possible, while failing at the end (long-circuit attack)

When a quantifier is found, max it out

When alternations are seen, pick the last one

For Example

(ab|cd|yz){1,20}g

abg

yzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyza

Not all Quantifiers are required

Consider (ab|cd|yz){1,20}g*
or (ab|cd|yz){1,20}g?
or even (ab|cd|yz){1,20}g{0,5}

yzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyzyza would match

All of This is a Challange to Automate...

Challange Accepted!

Before getting into benchmarking (the good stuff)

Let's look at what strings the automation produces

(Because it's unintentionally funny)

hi

ha

FYI: Green matches, Red does not...

ha

h1

a{1,15}h!

aaaaaaaaaaaaaaaha

(ab|cd|ef|gh|ij|kl|mn|op|qr|st|uv|wx|yz)d1

yzda

h+i

hhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhh
hhhhhhhhhhhhhhhhhhhh
hha

x(a{1,10}){1,10}

x1

x(a{1,10}){1,10}$

xaaaaaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaaa

aaaaaaaaaaaaaaaaa1

Benchrexes.pl

Becuase I can't think of a clever name for it

-Feed the script a txt file with regular expressions (1 per line)

-It will output a csv of benchmarks

-It benchmarks for both time and memory use

Results

Now lets look at some interesting real world results from

-Emerging Threats IDS rules
-regexlib.com

Tested in x64 Linux Mint 17 VM @ 1.7GHz (i7) with 1 GB of RAM

Regexlib.com/Memory

Most Complete URL Validator

```
^((http(s){0,1}\:V\V){0,1}([a-z|A-Z|0-9|\.|\-|_]){4,255}(\:\d{1,5})
{0,1}){0,1}((\V([a-z|A-Z|0-9|\.|\-|_]|\%[A-F|a-f|0-9]{2}){1,255})
{1,255}\V{0,1}){0,1}(|\V{0,1}\?[a-z|A-Z|0-9|\.|\-|_]{1,255}\=([a-
z|A-Z|0-9|\.|\-|_|\+|\:]|\%[A-F|a-f|0-9]{2}|\&[a-z|A-Z]{2,12}
\;){0,255}){0,1}((\&[a-z|A-Z|0-9|\.|\-|_]{1,255}\=([a-z|A-Z|0-9|
\.|\-|_|\+|\:]|\%[A-F|a-f|0-9]{2}|\&[a-z|A-Z]{2,12}\;){0,255})
{0,255})(\V{0,1}|\#([a-z|A-Z|0-9|\.|\-|_|\+|\:]|\%[A-F|a-f|0-9]{2}|\&
[a-z|A-Z]{2,12}\;){0,255})$
```

149 MB

Regexlib.com/Memory

Validates a long (windows) filename

^[^\\\./:*\?\"<>\|]{1}[^\\V:*\?\"<>\|]{0,254}$

660 KB

Regexlib.com/Time

Siteswap validator

```
^((0)*|(1|(2|(3|(4|50)0)0)0)*|(01*(2|(3|(4|50)0)0))*|(00(1|20)*(3|(4|50)0))*|(000(1|(2|30)0)*(4|50))*|(0000(1|(2|(3|40)0)0)*5)*|(2|(5(3|50)0|(4|51)
(2|40))0|(3|(4|51)1|(52|(4|51)50)0)(501|(4|5050)0)*(1|(3|50(2|40))0))*|(501|(4|5050)0|(1|(3|50(2|40))0)(2|(5(3|50)0|(4|51)(2|40))0)*(3|(4|51)1|
(52|(4|51)50)0))*|(0(2|(3|(4|51)1|(52|(4|51)50)0)(501|(4|5050)0)*1)*(5(3|50)0|(4|51)(2|40)|(3|(4|51)1|(52|(4|51)50)0)(501|(4|5050)0)*
(3|50(2|40))))*|(0(501)*(4|5050)|0(501)*(1|(3|50(2|40))0)(2|(5(3|50)0|(4|51)(2|40))0|(3|(4|51)1)(501)*(1|(3|50(2|40))0))*(52|(4|51)50|(3|(4|51)1)
(501)*(4|5050)))*|((01|0500)(40)*5|(0(2|40)0|(01|0500)(40)*(1|30))(2|(5(3|50)0|(4|51)(2|40))0|(3|(4|51)1|(52|(4|51)50)0)(40)*(1|30))*(3|(4|51)1|
(52|(4|51)50)0)(40)*5)*|((1|500)(40)*50|((2|40)0|(1|500)(40)*(1|30))(2|5(3|50)00|(3|520)(40)*(1|30))*(4|51|(3|520)(40)*50))*|(00(2|(4|51)20|
(3|(4|51)1|(52|(4|51)50)0)(501|(4|5050)0)*(1|(3|502)0))*(5(3|50)|(4|51)4|(3|(4|51)1|(52|(4|51)50)0)(501|(4|5050)0)*504))*|
(00(501|40)*505|00(501|40)*(1|(3|50(2|40))0)(2|(5(3|50)0|(4|51)(2|40))0|(3|(4|51)1|520)(501|40)*(1|(3|50(2|40))0))*((4|51)5|(3|(4|51)1|520)
(501|40)*505))*|((((3|50)0|1(2|40))0|(11|(2|150)0)(501|(4|5050)0)*(1|(3|50(2|40))0))(2|4(2|40)0|(3|41|4500)(501|(4|5050)0)*(1|(3|50(2|40))0))*5)*|
(000(2|(530|(4|51)(2|40))0|(3|(4|51)1|(52|(4|51)50)0)(501|(4|5050)0)*(1|(3|50(2|40))0))*55)*|(3|55(2|50)0|5(3|51)(1|40)|(4|52|5(3|51)50)
(51|(4|550)50)*(2|530|(4|550)(1|40)))*|(51|(4|550)50|(2|530|(4|550)(1|40))(3|55(2|50)0|5(3|51)(1|40))*(4|52|5(3|51)50))*|(50(51)*(4|550)|
(1|40|50(51)*(2|530))(3|55(2|50)0|(4|52)(51)*(2|530))*(5(3|51)|(4|52)(51)*(4|550)))*|(0(3|5(3|51)1|(4|52|5(3|51)50)(51|(4|550)50)*(2|(4|550)1))*
(55(2|50)|5(3|51)4|(4|52|5(3|51)50)(51|(4|550)50)*(53|(4|550)4)))*|(0(51)*(4|550)5|0(51)*(2|530|(4|550)(1|40))(3|55(2|50)0|5(3|51)(1|40)|(4|52)
(51)*(2|530|(4|550)(1|40)))*(5(3|51)5|(4|52)(51)*(4|550)5))*|((1|5050)(450)*5|(30|50(1|40)|(1|5050)(450)*(2|4(1|40)))(3|55(2|50)0|5(3|51)(1|40)|
(4|52|5(3|51)50)(450)*(2|4(1|40)))*(4|52|5(3|51)50)(450)*5)*|(050(51|450)*55|(0(1|40)|050(51|450)*(2|530|4(1|40)))(3|55(2|50)0|5(3|51)(1|40)|
(4|52|5(3|51)50)(51|450)*(2|530|4(1|40)))*(4|52|5(3|51)50)(51|450)*55)*|((5(2|50)0|(3|51)(1|40)|(2|(3|51)50)(51|(4|550)50)*(2|530|(4|550)(1|40)))
(3|4(51|(4|550)50)*(2|530|(4|550)(1|40)))*5)*|(((2|50)0|1(1|40)|150(51|(4|550)50)*(2|530|(4|550)(1|40)))(3|53(1|40)|(4|52|5350)(51|(4|550)50)*
(2|530|(4|550)(1|40)))*55)*|(00(3|5520|5(3|51)(1|40)|(4|52|5(3|51)50)(51|(4|550)50)*(2|530|(4|550)(1|40)))*555)*|(4|5(3|5(2|5(1|50))))*|
((3|5(2|5(1|50)))4*5)*|((2|5(1|50))(4|53)*55)*|((1|50)(4|5(3|52))*555)*|(0(4|5(3|5(2|51)))*5555)*|(5)*)$
```

.01 seconds

IDS Rule/Memory

GPL SQL time_zone buffer overflow attempt

TIME_ZONE\s*=\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))

325 KB

IDS Rule/Time

ET ACTIVEX ImageShack Toolbar Remote Code Execution

<object\s*[^>>]*\s*classid\s*=\s*[\x22\x27]\s*clsid\s*\x3a\s*{?\s*DC922B67-FF61-455E-9D79-959925B6695C\s*}?\s*(.*)\>

1.6 seconds

The Bad String

<object
^]]
classid =
"
clsid
:
{
DC922B67-FF61-455E-
9D79-959925B6695C }

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Real Bad Regex (Mem)

This slide not yet populated; it will be for the live presentation

Real Bad Regex (Time)

This slide not yet populated; it will be for the live presentation