ReScue: Crafting Regular Expression DoS Attacks (ASE'18)






Yuju Shen, **Yanyan Jiang**, Chang Xu, Xiaoxing Ma, Ping Yu, and Jian Lu

Nanjing University

System & Program Analysis Research (SPAR) Group

Regex, Regex Engine, and ReDoS

Regular Expressions (Regexes)

- The classical textbook definition
  - A single character is a regex
  - (Concatenation) if *A* and *B* are regexes, *AB* is also a regex
  - (Selection) if *A* and *B* are regexes, *A*|*B* is also a regex
  - (Kleene star, closure) if *A* is a regex, *A** is also a regex

- A powerful tool for pattern matching

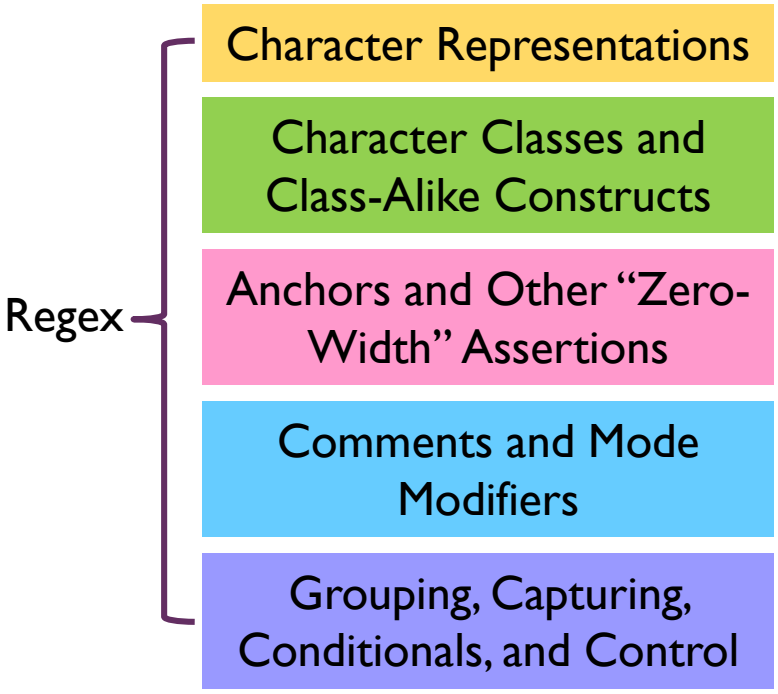
```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b
```

Regex Today: An Overly Powerful Tool for Pattern Matching

```
(?!^a?$|^(aa+)\1+$)
```

```
1 a
2 aa - Match
3 aaa - Match
4 aaaa
5 aaaaa - Match
6 aaaaaa
7 aaaaaaa - Match
8 aaaaaaaa
```

Regex Today: Features



\n	\t	\a	\b	\e	\f	\r	\v	\x{num}	
\xnum	\num	\unum	\Unum	\cchar	...				

[a-z]	[^a-z]	[[a-z]&&[^aei]]	\C	\X	\w				
\d	\s	\W	\D	\S	\p{Prop}	\P{Prop}			
[[:alpha:]]	[[.span-ll.]]	[[=n=]]	dot	...					

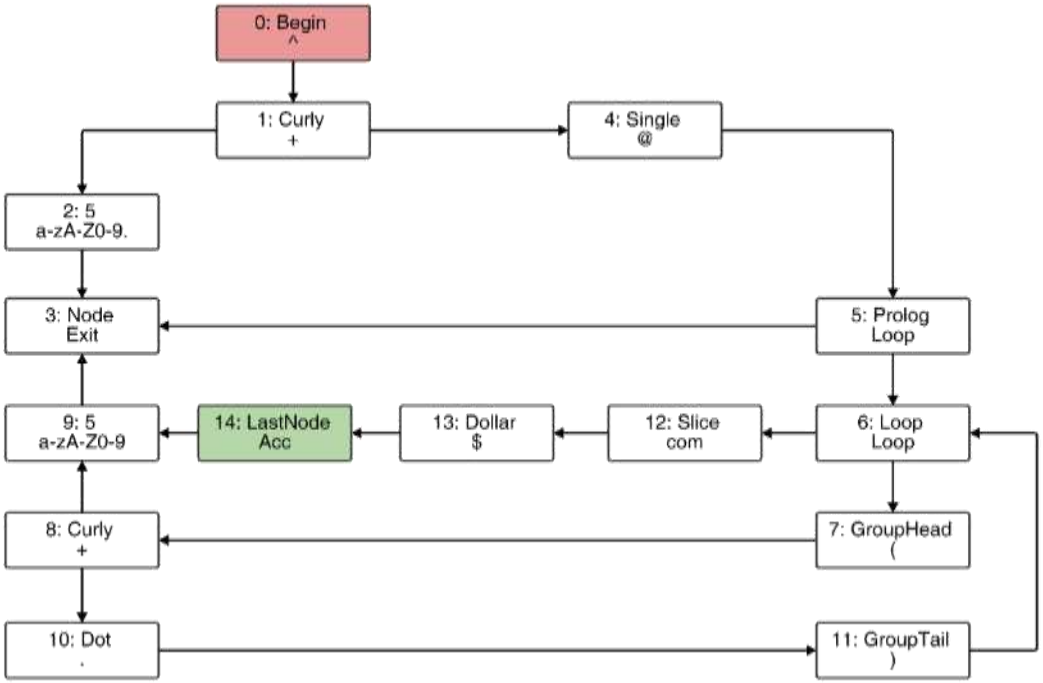
^	\A	$	\Z	\z	\G	\b	\B	\<	\>	...
(?=...)	(?!...)	(?<=...)	(?<!...)	...	...	...				

(?modifier)	(?i)	(?-i)	(?modifier:...)					(?:...)	
(?!...)	#...	\Q...\E	...	...	...	...	...	...	

(...)	\1	\2	(?:...)	(?<name>...)				(?>...)	
...|...|...	(? if then | else)			*	+	?	{m,n}		
*?	+?	??	{m,n}?	*+	++	?+	{m,n}+	...	

Regex Engine

- Well… these features are just too complicated
  - Let's do *backtracking search*! ← the root of all evil



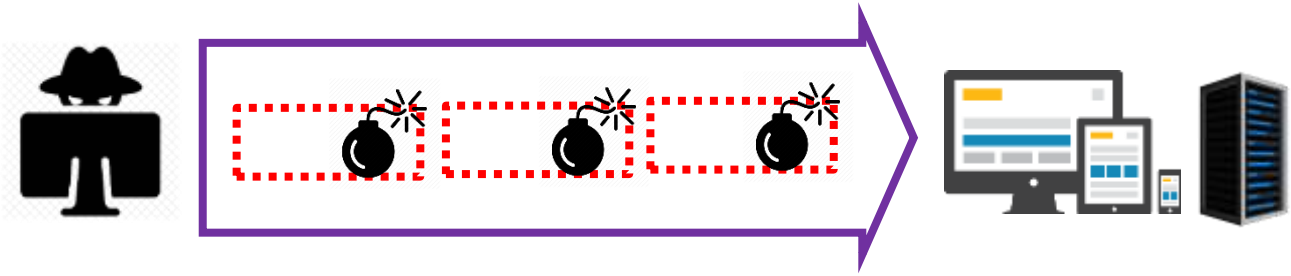
e-NFA of `^[a-zA-Z0-9._]+@([a-zA-Z0-9]+.)+com$`

ReDoS

- Backtrack search has an exponential worst case, and can be used to craft DoS attacks

```
^[a-zA-Z0-9._]+@([a-zA-Z0-9]+.)+com$
```

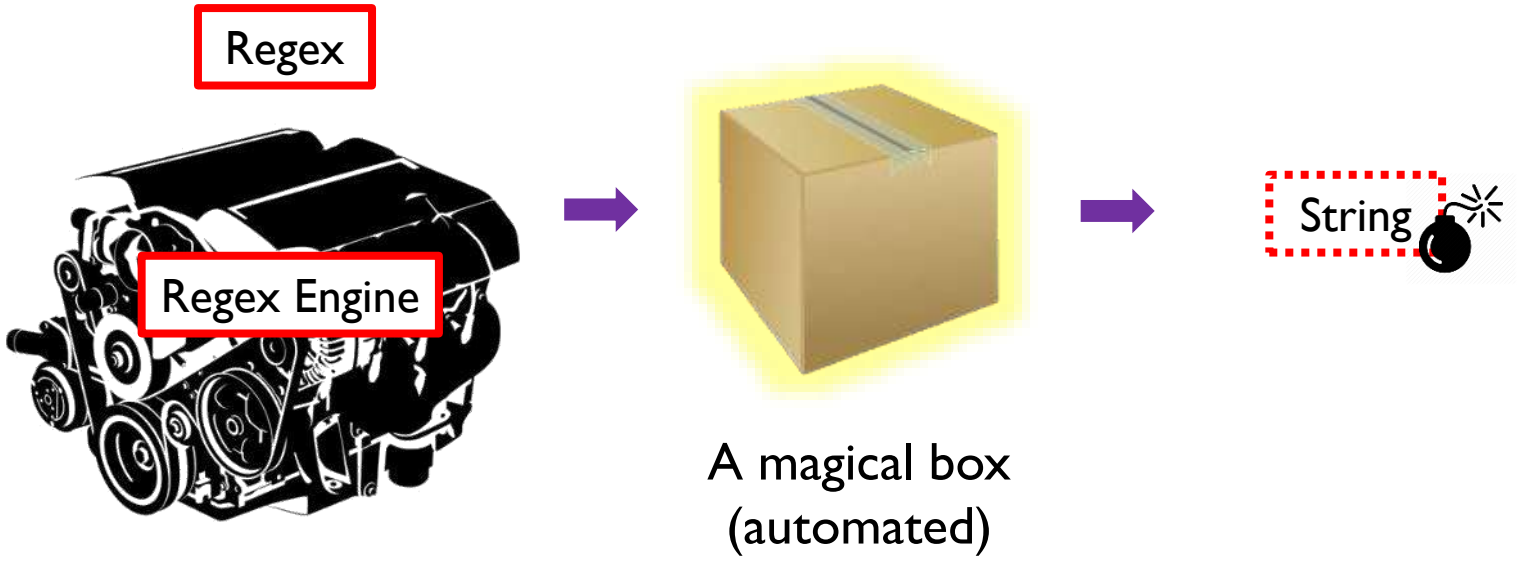
```
denny.syj@aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa                    0.014s
denny.syj@aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa     1.056s
denny.syj@aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  31.247s
```



Automatic ReDoS Detection

The ReDoS Detection Problem

- Find a string for a regex, which can cause a timeout matching on its matching engine



Analyze the Example

```
^[a-zA-Z0-9._]+@([a-zA-Z0-9]+.)+com$
```

“denny.syj@aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa”

Wanted

Maximized matching time
Length no more than ***n***

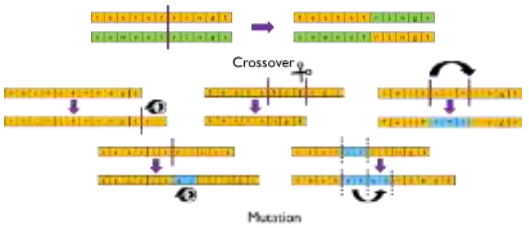
Number of all possible strings = $|\Sigma|^n$

- ReDoS detection is a huge-space search problem

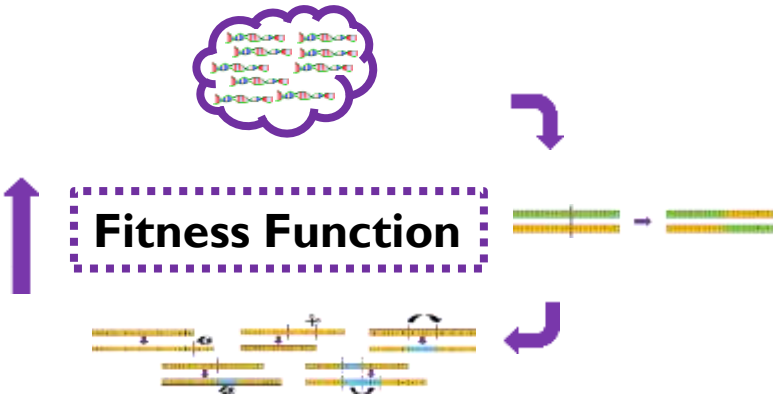
An Intuitive Solution: Genetic Algorithm

- Genetic representation
  - Individual
  - Population

- Genetic operations
  - Crossover
  - Mutation

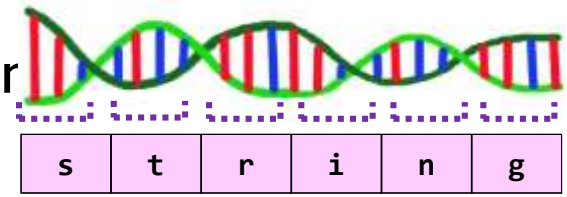


- Selection Strategy
  - Fitness function



An Intuitive Solution: Genetic Algorithm

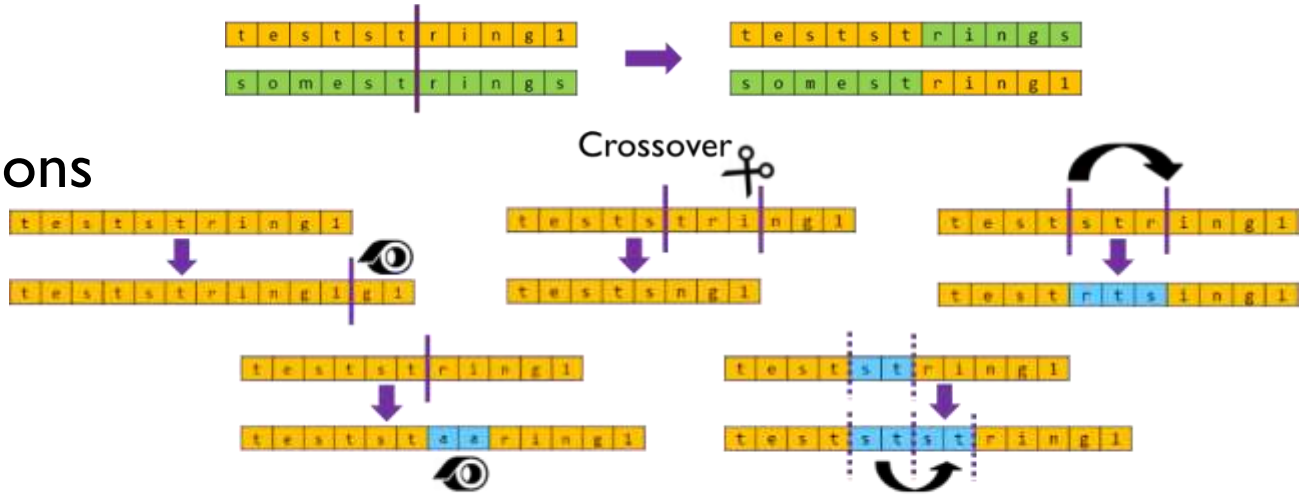
- Genetic representation
  - Individual
  - Population



Individual


Population

- Genetic operations
  - Crossover
  - Mutation



Mutation

- Selection Strategy
  - Fitness function

$$f(s) = \frac{|matching\ time|}{|s| + 1}$$

So Far, So Good.

The Simple GA Can't Solve Many Problems!

The poorly designed part

```
(0|[0-1]){2,15}(hello)\2([0-9]+)+#
```

- The attack string we want cannot be generated by GA

Long prefix

```
00hellohello4215432421328179574 81...
```

The Dilemma Between Attack String and Long Prefix

- The fitness function

$$f(s) = \frac{|matching\ time|}{|s| + 1}$$

The **slower** the better

The **shorter** the better

"00hellohello"

Fast

Long

- GA tends to generate short but time-consuming strings, but prefixes like "**00hellohello**" is long and not time-consuming, so it is hard to be generated and kept in the population

ReScue: A Cleverer Algorithm

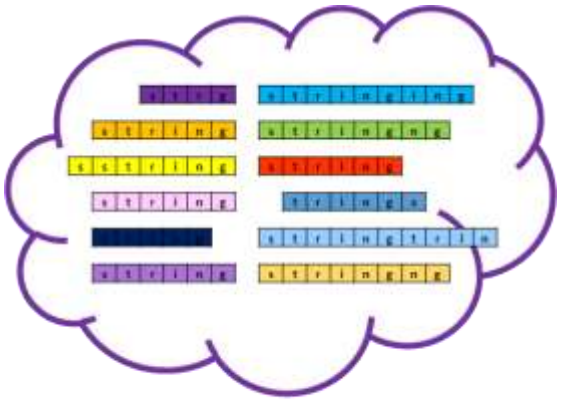
The Dilemma between Attack String and Long Prefix

$$f(s) = \frac{|matching\ time|}{|s| + 1}$$

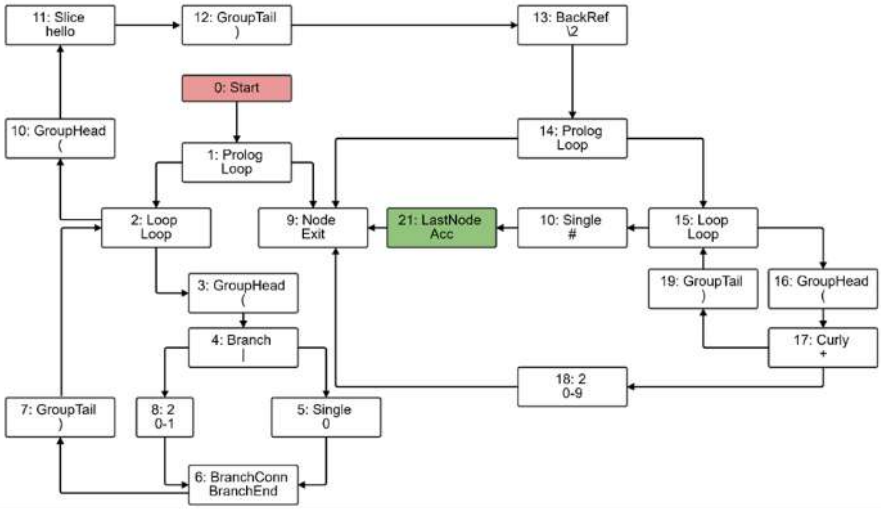
The slower the better

The shorter the better

"00hellohello" Fast Long



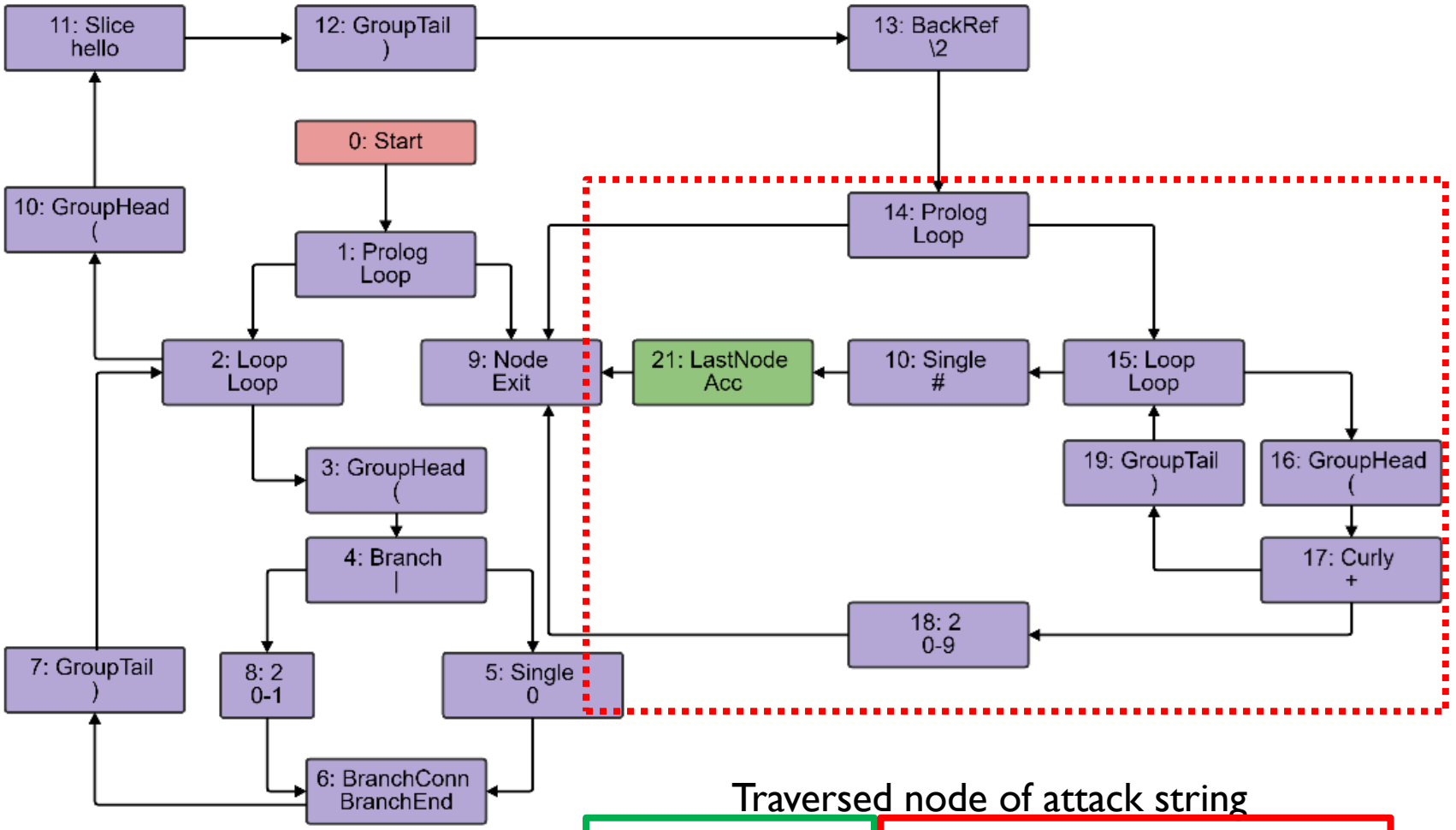
Population



00hellohello42154324213281795748 1...

e-NFA of (0|[0-1]){2,15}(hello)\2([0-9]+)+#

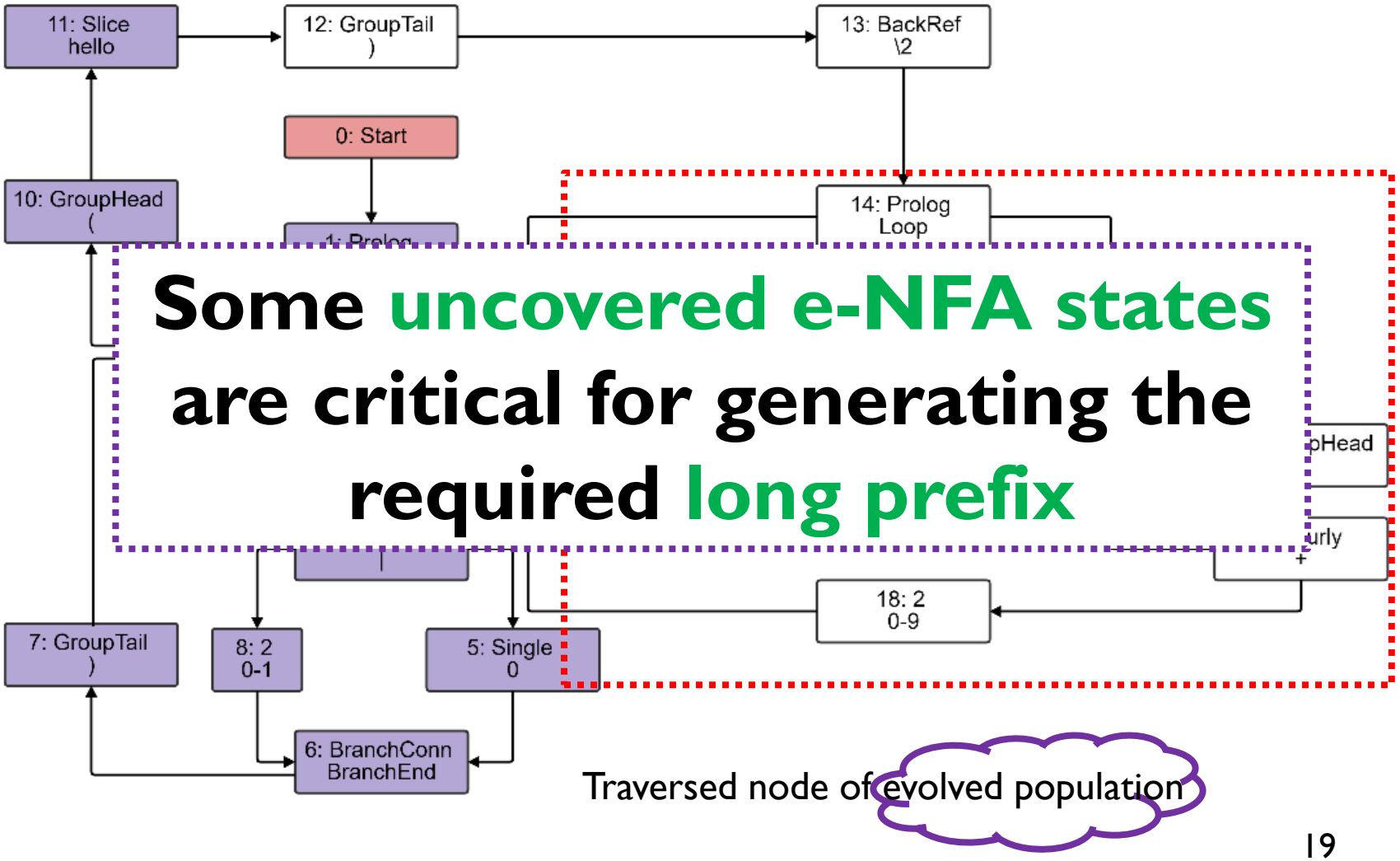
A Successful Attack String



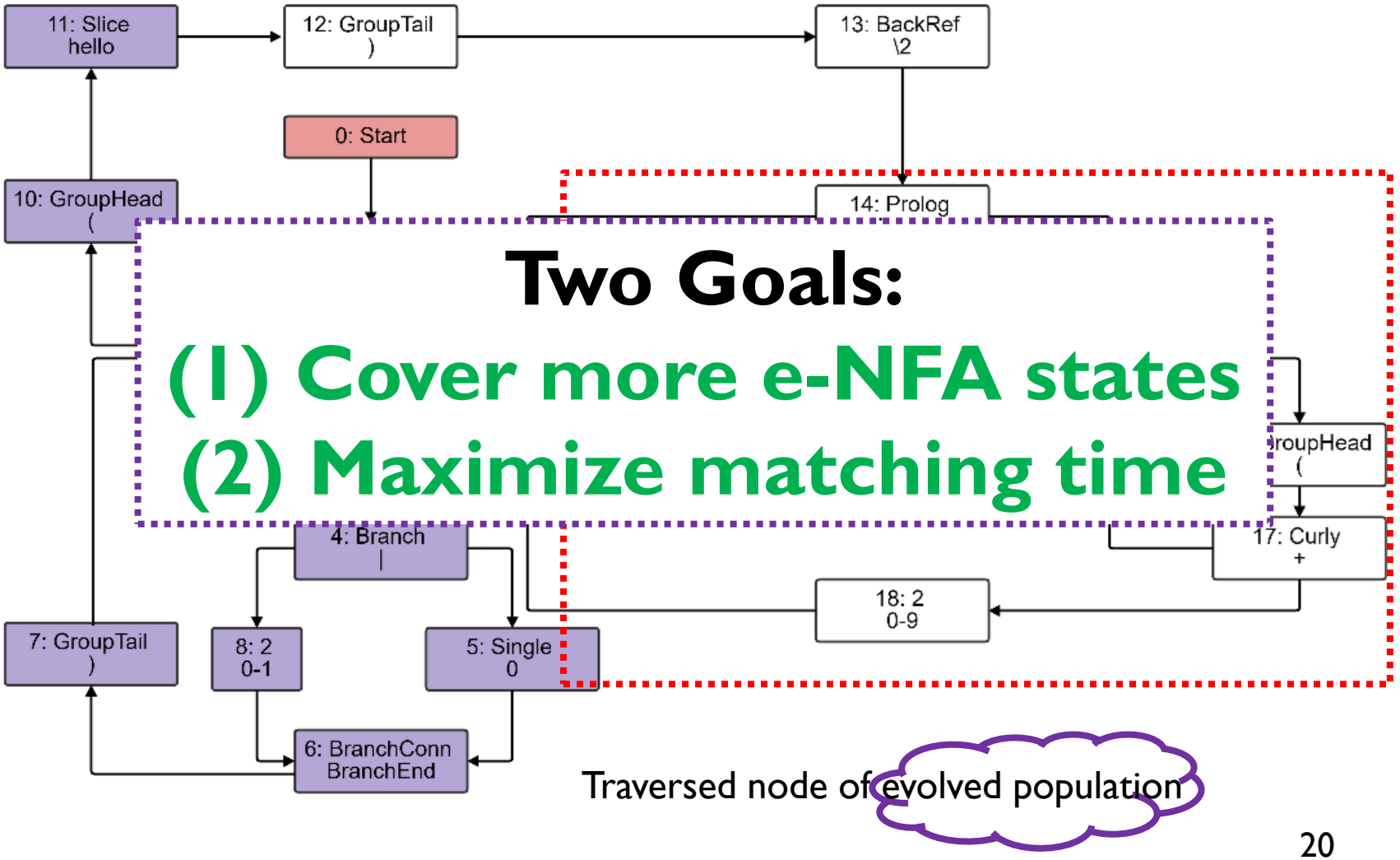
Traversed node of attack string

`00hellohello42154324213281795748 1…`

Attack Candidate by the Naïve GA



Our Solution



Ingredient #1 (Seeding): Cover More e-NFA States

- Search a string set that cover as many e-NFA states as possible
  - GA
  - Fitness function

$$f = \begin{cases} 1, & \text{Cover at least one new e-NFA State} \\ 0, & \text{Otherwise} \end{cases}$$

Ingredient #2 (Incubating): Maximize Cost-Effectiveness

- Search strings that match slowly
  - GA
  - Inherit the high coverage string set from the seeding phase and preserve its coverage
  - Fitness function

$$f(s) = \frac{|matching\ time|}{|s| + 1}$$

Ingredient #3 (Pumping): Exploit the Pumping Lemma

- (Pumping Lemma) For a sufficiently long string *w* in any regular language *L*, it can be written as *w* = *xyz* where {*xz*, *xyz*, *xyyz*, *xyyyz*, xyyyyz, …} ⊆ *L*

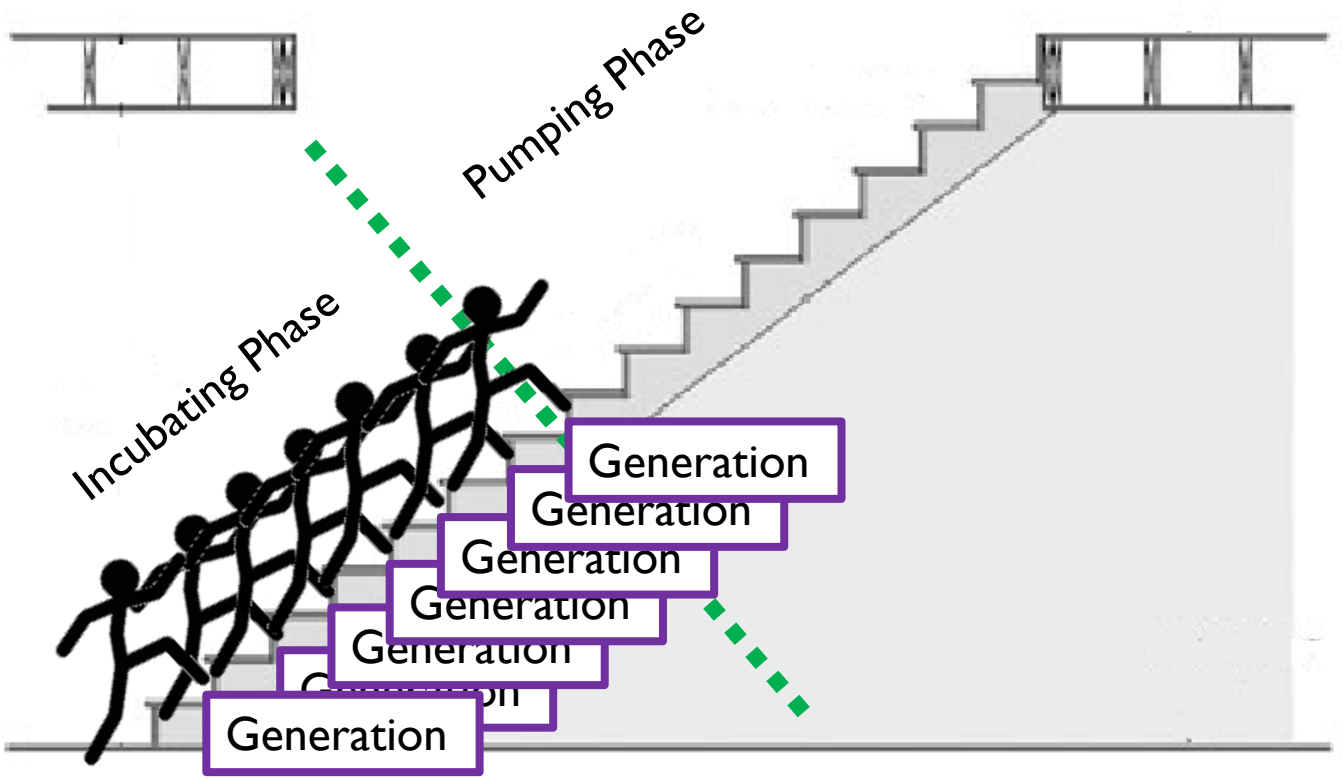
- Even if today's regexes do not correspond to regular language, we can still *pump* them!

Slow (found by GP)

Extremely slow
(enhanced by pumping)

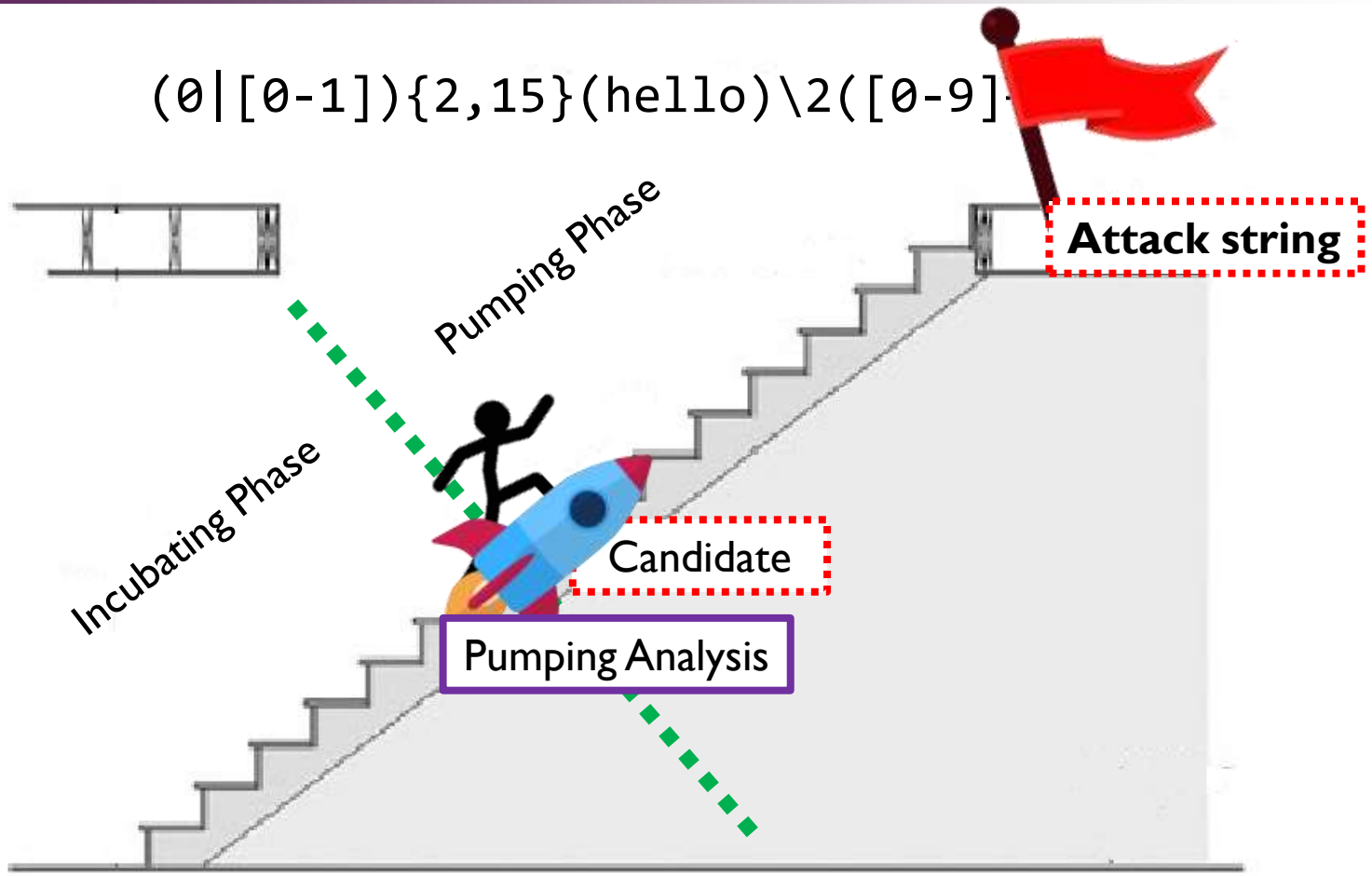
Terminate the Incubating Phase Early

```
(0|[0-1]){2,15}(hello)\2([0-9]+)+#
```



- The incubating phase generates an intermediate population, with contains at least a slow-matching candidate

Pumping Phase



- The pumping phase directly generate the final attack string from the candidate of the intermediate population

Putting Things Together

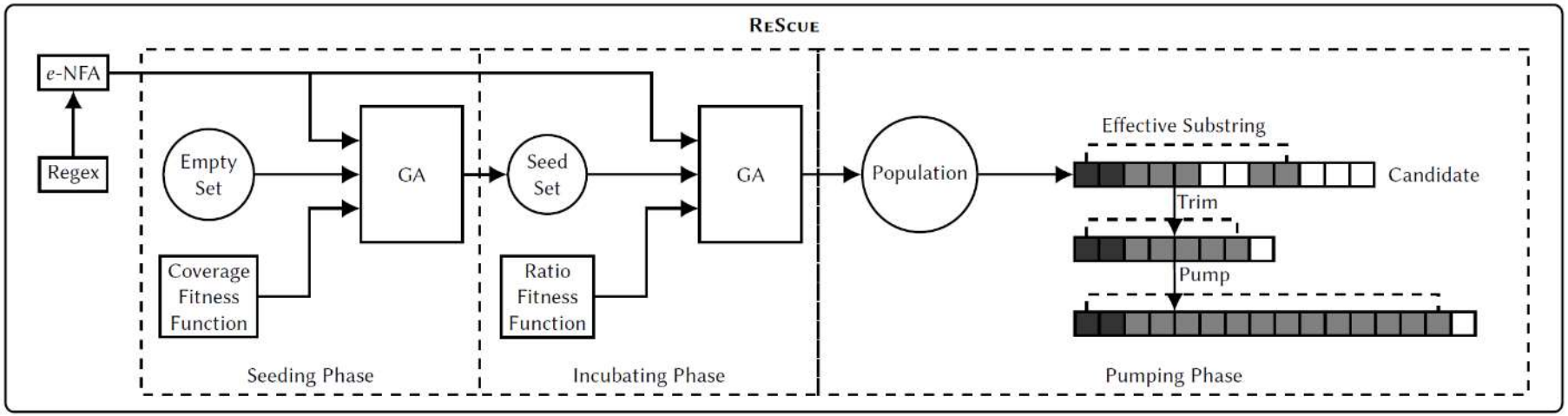


Figure 1: Overview of the ReScue technique for automated ReDoS string generation.

- Seeding Phase: Cover as many e-NFA states as possible
- Incubating Phase: Generate slow-matching candidates
- Pumping Phase: Enhancing the candidates to attack strings

Evaluation

Evaluation

- Compare ReScue with existing tools
  - Effectiveness
  - Efficiency
- Apply ReScue on github projects to detect real-world ReDoS vulnerabilities

Evaluation

- Data set
  - RegLib, Snort [Rathnayake et al. '14]
  - Corpus [Chapman et al. ISSTA'16]

Name	Number
RegLib	2, 992
Snort	12, 499
Corpus	13, 597
Total	29, 088

- Comparison
  - **ReScue**
  - SlowFuzz [Petsios et al. CCS'17]
  - RXXR2 [Rathnayake et al. '14]
  - Rexploiter [Wüstholz et al. TACAS'17]
  - NFAA [Weideman et al. CIAA'16]

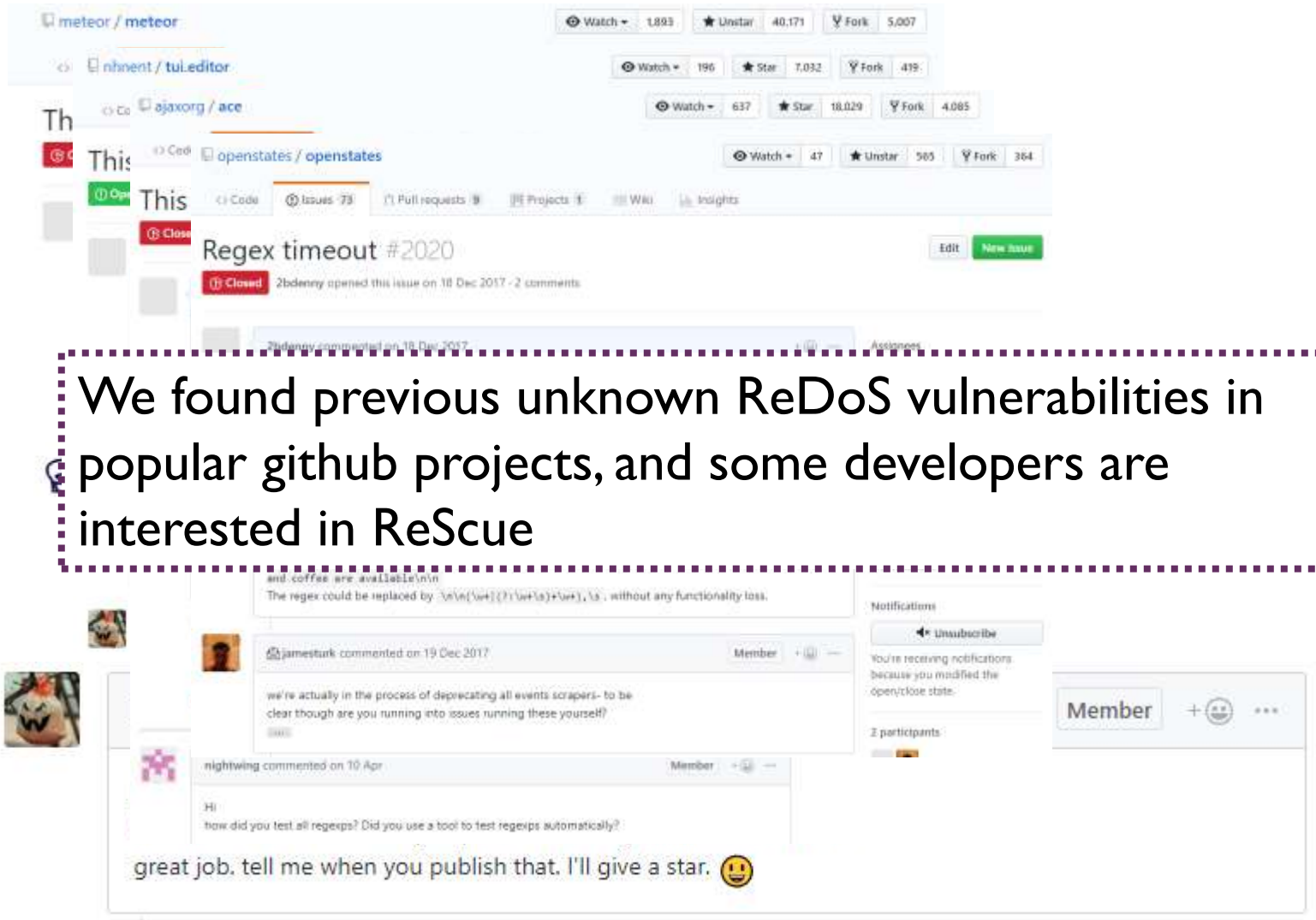
Evaluation Result

- Compared to existing tools

Tool	Detected ReSoS	FP	TP Rate	Average Time(s)
ReScue	**186 (82%)**	**-**		**0.6128**
SlowFuzz (+84%)	101 (44%)	-		0.5965
RXXR2 (+49%)	125 (55%)	80	61%	0.0025
Rexploiter (+520%)	30 (13%)	2152	1.30%	0.4073
NFAA (N/A)	0 (0%)	714	N/A	2.1546
Summary	227 (100%)			

- ReScue can detect at least 49% more ReDoS than existing tools, and the detection time is reasonable

Real-World ReDoS Vulnerabilities



We found previous unknown ReDoS vulnerabilities in popular github projects, and some developers are interested in ReScue

Stories Behind the Scene

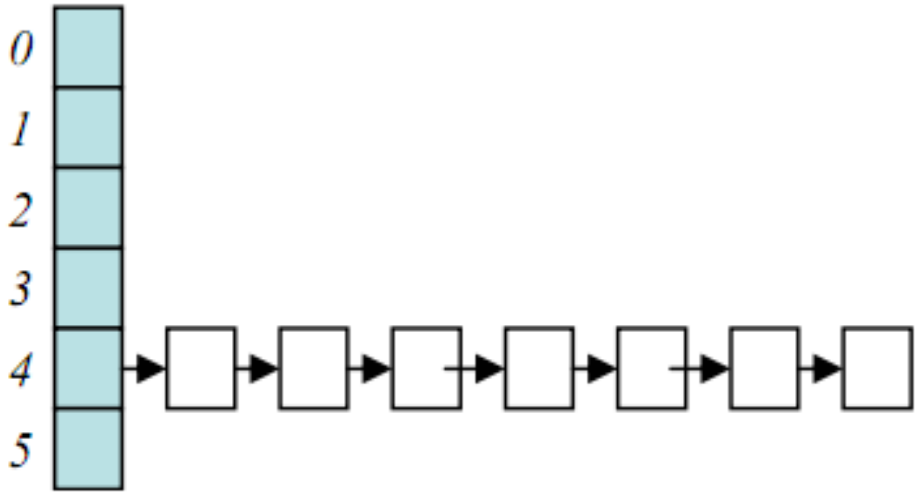
Stories Behind the Scene

- Why we're doing security stuffs?
  - I don't know!

- Why we're doing algorithmic complexity attack stuffs?
  - Just a few words from Tim Roughgarden's lecture
  - Points to a USENIX Security 03 paper by Scott A. Crosby and Dan S. Wallach
    - "Denial of Service via Algorithmic Complexity Attacks"

A (Not-Yet-Dead) Dead End

- Our first attempt is to hack hash tables
  - In Java world, (*A*|*B*)* hashes to zero if *A*.hashcode() = *B*.hashcode() = 0



- This attempt fails for Java 8 (this issue is *permanently* fixed)

Hmm, Regexes…

- Regex is a much better target for algorithmic complexity attacks!
  - They are widely used
  - They have *exponential* worst case
  - And developers simply do not write regexes with care

- Bang~

Thank You!

https://*2bdenny*.github.io/ReScue

Nanjing University

System & Program Analysis Research (SPAR) Group