Regular expression attack technique on ReDoS vulnerability

DOI: https://doi.org/10.54654/isj.v1i21.1030

Nguyen Trung Dung, Pham Van Toi, Phung Minh Hieu

***Abstract*— Regular expressions, or regexes, have become an integral part of modern software development, seamlessly woven into the fabric of countless applications. From validating user input in web forms to parsing complex log files for data analysis, regexes are employed across a vast spectrum of tasks. Their ability to precisely define and match patterns within text makes them invaluable tools for tasks ranging from simple data extraction to sophisticated security measures. However, this widespread reliance on regexes also introduces a significant security vulnerability: ReDoS (Regular Expression Denial of Service) attacks. These attacks exploit the inherent complexity of regex matching by crafting malicious input that triggers an exponentially long processing time, effectively bringing the application to a standstill. The potential for ReDoS attacks highlights the crucial need for developers to exercise extreme caution when designing and implementing regex-based components within their applications. This paper explores the inherent ambiguity of regular expressions, fuzzing with static analysis and proposes a novel fuzzing technique to generate effective attack patterns. By analyzing the potential interpretations of ambiguous regex constructs, our method identifies and exploits weaknesses in software implementations that rely on regex for input validation. The proposed fuzzing algorithm generates test cases that systematically explore the ambiguity space, maximizing the likelihood of uncovering vulnerabilities related to unexpected regex behavior. This approach aims to enhance software security by proactively detecting and mitigating potential attack vectors stemming from the misinterpretation of regular expression patterns.**

This manuscript is received on April 16, 2024. It is commented on May 22, 2024 and is accepted on June 10, 2024 by the first reviewer. It is commented on May 21, 2024 and is accepted on June 14, 2024 by the second reviewer.

***Tóm tắt*— Biểu thức chính quy (regex) là một phần không thể thiếu trong phát triển ứng dụng phần mềm, được tích hợp một cách liền mạch vào vô số ứng dụng. Biểu thức chính quy được sử dụng trong nhiều nhiệm vụ khác nhau từ xác thực đầu vào của người dùng ứng dụng web đến phân tích dữ liệu văn bản phức tạp. Tuy nhiên, sự phụ thuộc vào biểu thức chính quy cũng mang đến một lỗ hổng bảo mật nghiêm trọng như tấn công ReDoS (Regular Expression Denial of Service - Tấn công từ chối dịch vụ trên biểu thức chính quy). Tấn công ReDoS khai thác dựa trên sự phức tạp của việc so khớp biểu thức chính quy bằng cách tạo ra đầu vào độc hại để thời gian xử lý tăng theo cấp số mũ, khiến ứng dụng ngừng hoạt động. Mục đích của các cuộc tấn công ReDoS tập trung vào các nhà phát triển phần mềm khi thiết kế và triển khai các thành phần dựa trên biểu thức chính quy trong sản phẩm. Bài báo bao gồm thuật toán ambiguity của biểu thức chính quy, kỹ thuật fuzzing kết hợp với phân tích tĩnh, từ đó đề xuất một kỹ thuật mới để tạo ra các mẫu tấn công hiệu quả. Bằng cách phân tích cấu trúc ambiguity, nhóm tác giả đã xác định phương pháp và kỹ thuật khai thác điểm yếu trong cách triển khai phần mềm dựa vào biểu thức chính quy để xác thực đầu vào. Thuật toán đề xuất được thực nghiệm một cách có hệ thống, tối ưu hóa khả năng phát hiện các lỗ hổng liên quan đến hành vi trên biểu thức chính quy. Cách tiếp cận của bài báo nhằm tăng cường bảo mật phần mềm thông qua việc chủ động phát hiện và giảm thiểu các vectơ tấn công tiềm ẩn phát sinh từ việc hiểu sai cách sử dụng biểu thức chính quy.**

Keywords*— regex; fuzzing; automaton; ambiguity; ReDoS.

Từ khóa*— biểu thức chính quy; kỹ thuật fuzzing; automaton; ambiguity; lỗ hổng ReDoS.

I. INTRODUCTION

Regular expressions (regex) are a powerful tool used for searching and manipulating text. They allow you to describe complex text patterns and use them to validate data, extract information, or replace text. However, regex can also be a source of security vulnerabilities, particularly when used carelessly. One of the most common vulnerabilities is ReDoS (Regular Expression Denial of Service) [1, 2]. ReDoS occurs when a poorly designed regex has to process an input string that can trigger backtracking [3] when the regex engine tries various combinations to match the input string. In some cases, this backtracking can become extremely lengthy, leading to the consumption of significant system resources and causing the server to hang or become sluggish.

ReDoS attacks typically target web applications where users can provide input to form fields or URL parameters. An attacker can craft a malicious input [4] that triggers backtracking in the application's regex, resulting in a denial-of-service for legitimate users. This vulnerability has been demonstrated in various platforms, including .NET [5], where a single malicious input could paralyze applications.

Preventing ReDoS attacks is a challenge, developers often struggle to identify vulnerable regexes and rewrite them to avoid exponential complexity. Additionally, implementing effective sanitizers to filter out malicious input requires understanding the intricate patterns that trigger worst-case behavior. While numerous research efforts have been dedicated to the detection and prevention of ReDoS attacks, a comprehensive understanding of attack chains that leverage ReDoS vulnerability discovery mechanisms remains unexplored. This paper leverages a combination of vulnerability analysis and detection techniques to construct a comprehensive attack chain specifically targeting ReDoS vulnerabilities. By meticulously analyzing the mechanics of ReDoS, we identify exploitable weaknesses and develop a methodology for crafting malicious input strings that trigger excessive backtracking in regex engines. This allows us to systematically evaluate the effectiveness of existing ReDoS detection mechanisms and propose novel countermeasures to mitigate the risks associated with this vulnerability.

The contributions of this paper are summarized in three main points:

- *Exploring algorithms related to Regex and ReDoS attacks:* Show the ambiguity inherent in regular expressions (Regex) to understand how they can be interpreted in multiple way; Exponential Degree Ambiguity [6]: Analyze the types of ambiguities in Regex that lead to exponential complexity in the Regex algorithm, which can be exploited for ReDoS attacks. Infinite Degree Ambiguity [6]: Investigate cases where Regex has infinite ambiguity, making it more difficult to analyze and predict the behavior of the Regex. Fuzzing with Static Analysis: Combine fuzzing with static analysis to generate input data strings that can trigger unexpected behavior in Regex; Apply the selective memorization technique developed by Davis et al. [7] to optimize Regex algorithm performance, particularly when handling large datasets.

- *Introduction a string generation algorithm for attacks:* The paper introduces the new algorithm for generating attack string based on automata theory and fuzzing algorithm.

- *Implementation and evaluation:* The paper details the implementation of the proposed algorithm and evaluates its performance on a dataset.

II. Related Work

A regular expression is a text processing utility for programmers. Regular expressions are widely used in all kind of software, inside a lot of libraries from program languages. Since regular expressions are easy to get wrong [8], which may help attackers to bypass checks [9]. There are two ways to implement regular expression matching. One uses deterministic finite automaton (DFA), and another uses backtracking. Recent implementations like Go's regexp or Rust's regex use DFA like approach. However, implementations on many programming languages use the backtracking approach.

When the regular expression matching is implemented based on the backtracking, it may take exponential or superlinear time complexity against an input string length. In other words, a short string may invoke a long matching time to a regular expression. For example, the relation between matching times against /^(a|a)*$/ and input string lengths, shows extremely increasing matching time against input string length (Figure 1).

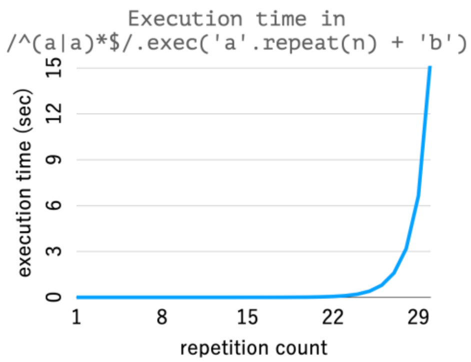


Figure 1. Regular Expression's execution time

Ambiguity of Regular Expression

A regular expression is ambiguous if there are multiple matching processes for one string. For example, /^(a|ab)(bc|c)$/ is ambiguous because there are two matching processes for the string 'abc'.

A regular expression is ambiguous does not immediately mean that it is ReDoS vulnerable. In the previous example, it is obvious that the matching time will not explode because it has only finite ambiguity. However, there is a deep relationship between ReDoS vulnerability and ambiguity of regular expression.

We show another example about an ambiguous regular expression with a repetition quantifier /^((a|ab)(bc|c))*$/. Specifically, giving the string 'abc'.repeat(30) + 'a' can invoke a very long matching time. In fact, the matching time complexity of the regular expression is exponential. The regular expression is ReDoS vulnerable.

A regular expression (regex) is vulnerable to ReDoS (Regular Expression Denial of Service) attacks when it exhibits partial infinite ambiguity. This ambiguity arises from the presence of repetition constructs within the regex, allowing for repeated matching of certain sub-patterns. The combination of repetition and ambiguity creates a scenario where the engine can become trapped in an iterative backtracking process, attempting to explore all possible interpretations. As the backtracking process continues without reaching a conclusive match, the engine consumes an excessive amount of computational resources, ultimately leading to a denial of service condition.

EDA and IDA

We will proceed with NFA which is equivalent to the regular expression. However, we assume that the NFA is converted to reflect the regular expression structure exactly (the NFA constructed by Thompson construction [10] without any determinization or minimization).

The ambiguity of a regular expression means that there are multiple ways to transition from one state to another in a given string on the NFA.

Suppose that the ambiguous transition is in a loop of the NFA transition diagram. In this case, there are two transitions to return to the same state with the same string $w (Figure 2). This means that the regular expression has infinite ambiguity. The reason is that for a string w^n with w in n order, there are 2^n ways to transition, depending on whether it chooses the above or the bottom transition for each w.

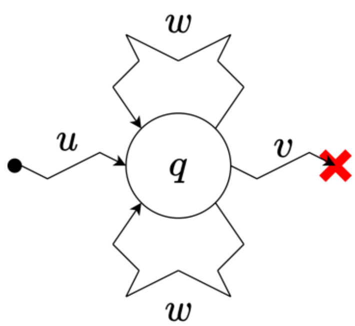


Figure 2. The Transition of EDA Structure

A structure in the transition diagram of NFA is called EDA (Exponential Degree Ambiguity) structure [6], and it is the cause of cases where the matching time becomes exponential.

It is not only in the presence of EDA that regular expressions have infinite ambiguity. Suppose that there is an ambiguous transition across two loops (Figure 3), the same string w can transition around the first loop and the next loop, and can also transition between the first states of the two loops.

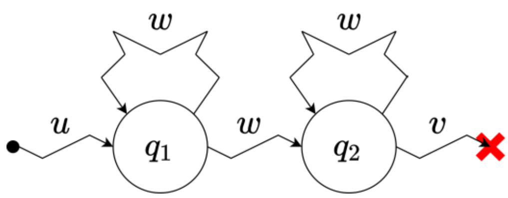


Figure 3. The Transition of IDA Structure

The string w^n, there are n transitions between the two loops with w, so there are n different ways to transition. This also means that the regular expression has infinite ambiguity.

A structure in the transition diagram of NFA is called IDA (Infinite Degree Ambiguity) structure [6]. It is the cause of cases where the matching time is polynomial of the second or higher degree.

Automated Detection of ReDoS

Regular expressions (regexes) are often poorly tested and vulnerable to ReDoS attacks, which can cause denial of service by exploiting inefficient matching algorithms. While some regex engines offer limited mitigation through matching limits, these are not widely used and ineffective against attacks flooding a server with malicious regexes. To address this, automated tools for detecting ReDoS vulnerabilities are crucial. These tools can be static, analyze the regex structure for potential issues, or dynamic, actually running matches to identify problematic strings. Static analysis can have false positives and miss vulnerabilities related to backreferences [11]. Dynamic analysis is more accurate, but it is time-consuming and may miss vulnerabilities requiring specific input sequences. The ideal solution would be a comprehensive tool that combines both approaches to effectively identify and mitigate ReDoS vulnerabilities in regexes.

Fuzzing

Fuzzing is an automated software testing method that generates a large amount of input called "fuzz" and actually gives it to a program to check if it shows any problematic behavior (i.e. bug). In contrast to static analysis, where the program is not actually run, fuzzing can be considered as a kind of dynamic analysis, where the program is actually run.

One of the features of fuzzing is that evolutionary computation methods such as genetic algorithms are sometimes used to generate fuzz in order to find bugs efficiently.

There is previous research by Shen et al [12]. that used fuzzing to detect ReDoS. In Shen et al.'s research, fuzzing was performed in the following three steps. The implementation of algorithm follows the same general flow, but the details are completely different.

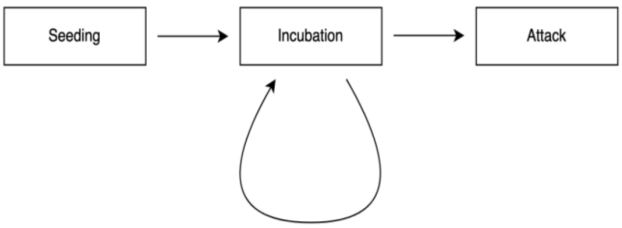


Figure 4. Fuzzing process to detect ReDoS

- Seeding: the step where the initial generation of the genetic algorithm is determined from the given regular expression.
- Incubation: the step of iterating through the generations of the genetic algorithm to produce a string that takes more matching time.
- Attack: the step where the strings generated by incubation are used to test whether the matching time is high enough even under conditions close to those of the real attack.

(1) Seeding is the step where the given regular expression is statically analyzed to obtain the initial generation of the genetic algorithm.

Let's observe the EDA and IDA structures. You will see that the EDA structure and the IDA structure have the following pair of states (q_1, q_2) in common.

There are two different transitions from q_1 with the same letter a.

- q_1 and q_2 can be transitioned with the same letter b.

- q_2 can transition with the letter a.

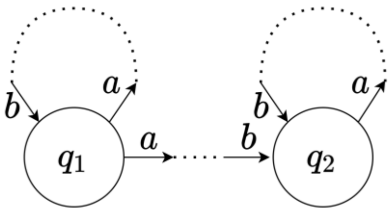


Figure 5. Seeding Phase

In a way, this pair of states is like a seed of IDA or EDA structure. Adding a repeated string on the string between these two states to the initial generation, we expect the fuzzing to efficiently increase the matching time.

The idea of statically analyzing regular expressions to obtain strings that may cause ReDoS vulnerabilities can be seen in the research of Li et al [5]. However, that work was the syntax direction analysis of regular expressions, and there was a possibility of missing parts of EDA or IDA structures. The observation of state pairs is similar to that of Linear Time Property in Chida et al.'s research [13], but this observation is more detailed.

(2) Incubation is the step in which the initial generation obtained by seeding is turned into a string that takes longer to match using a genetic algorithm.

If we find a string that takes enough matching time at this stage, we move on to attack to verify if the string is attackable. In order to acceleration regular expression matching, which will be explained on the next page, the number of times a character is read during the matching is used to determine if the matching takes time or not, rather than the actual matching time.

If no attackable string is found after repeating the generation of the genetic algorithm for the specified number of times, we report that the regular expression is safe.

In this paper, the genetic algorithm uses strings with repetition structures instead of normal strings as genes. A string with repetition structures is a sequence of a normal string w and a string to be repeated $(w)^n$, and the number of repetitions can be changed from outside. This is a genetic programming approach to make a string with structures that can be changed by variables in a gene. In addition, repetition structures are actually encoded in the sequence, which makes mutation and crossover in genetic algorithms efficient.

(3) Attack is the step to verify whether an attack is actually possible by matching the strings found in incubation while adjusting the number of iterations.

As mentioned above, the genetic algorithm uses strings with repetition structures, and we apply this to determine whether the matching time is exponential or polynomial.

First, assuming that the matching time increases exponentially, we try to match the number of repetitions as the logarithm of the threshold. If the threshold is reached here, the matching time is exponential. If not, assume that the matching time is polynomial increasing of order d and do the same. Repeating this until d becomes 2, and if the threshold is not reached until the end, we assume that it is safe for this string and return to incubation.

Detecting ReDoS Tools

ReScue [12], SlowFuzz [14], RXXR2 [15], Rexploiter [16], and NFAA [17] represent significant advancements in the research of mitigating Regular Expression Denial of Service (ReDoS) vulnerabilities. ReScue applies static analysis methodologies to systematically detect regex patterns susceptible to excessive backtracking, aiming to preemptively identify and correct inefficiencies before deployment. SlowFuzz adopts a dynamic approach by employing fuzz testing, generating extensive input variations to empirically uncover performance bottlenecks and potential vulnerabilities within regex engines. RXXR2 integrates empirical testing and static analysis to deliver a comprehensive framework for identifying regex vulnerabilities, enhancing detection precision through rigorous evaluation. Rexploiter emphasizes the generation of exploitative attack strings, providing a practical lens through the impact of identified vulnerabilities can be assessed. NFAA (Nondeterministic Finite Automata Analyzer), utilizes principles from automata theory to analyze the computational complexity of regex patterns, ensuring they are devoid of performance-degrading constructs. Collectively, these tools not only advance the detection and mitigation of ReDoS attacks but also contribute to the broader field of software security by integrating static and dynamic analysis techniques with theoretical foundations, offering a multifaceted approach to safeguarding regex implementations. We will use these tools for our research.

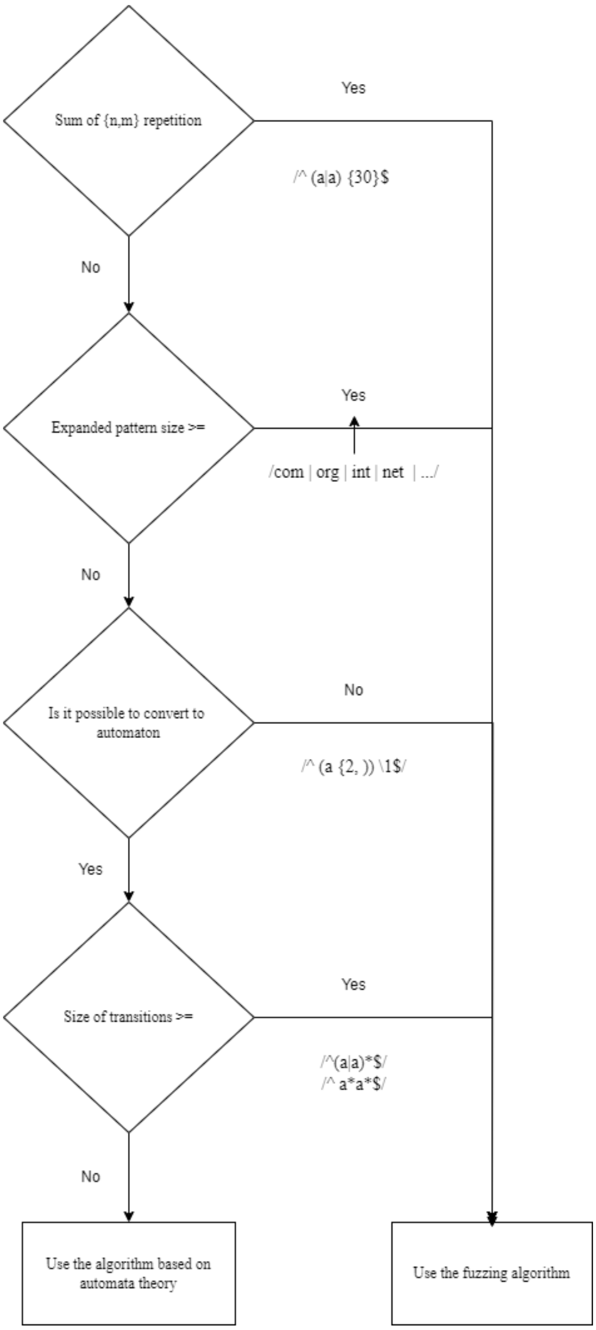


Figure 6. Choosing techniques based on detection

III. Attack algorithm

This paper introduces a algorithm to check ReDoS vulnerability in the given regular expression. You can find vulnerabilities in the given regular expression and can obtain an attack string to the vulnerability.

The ideal approach to regular expression vulnerability detection involves a hybrid strategy combining the strengths of both automata-based [18] and fuzzing algorithms [12].

TABLE 1: COMPARATIONS BETWEEN AUTOMATON AND FUZZING [11, 18]

	Automata-based	**Fuzzing**
Pros	- Fast detection (no actual matching). - Theoretically accurate detection.	- Handles all practical regular expressions. - Practical vulnerability detection (not just theoretical)
Cons	- State explosion can make detection slow. - Not all practical regular expressions can be handled	- Can erroneously detect vulnerable expressions as safe - Detection takes time due to actual matching

First algorithm represents an attack pattern, which is a string composed of fixed and repeating parts. The repeating parts can be repeated a variable number of times, determined by a complexity level and a limit. The algorithm ensures the adjusted count does not exceed the maximum size limit for the generated string. Finally, the algorithm provides methods to return the string representation of the attack pattern in various formats, allowing for different styles based on the desired output (e.g., JavaScript, Python). The adjusted repetition count and the resulting string representation make this algorithm suitable for generating attack patterns with variable lengths and complexity levels.

Algorithm 1: Generating Attack Pattern

Input: A pump, a suffix and a count for the pumps n
Output: An attack string pattern
begin

```
  function calculateFixedSize(pumps):
  fixedSize = 0
```

```
  for pump in pumps:
    fixedSize = fixedSize + pump.firstPart.sizeAsString
  fixedSize = fixedSize + suffix
  return fixedSize

function calculateRepeatSize(pumps):
  repeatSize = 0
  for pump in pumps:
    repeatSize = repeatSize + pump.secondPart.sizeAsString
  return repeatSize

function attackPattern(complexity, limit, maxSize):
  // Calculate fixed and repeat sizes
  fixedSize = calculateFixedSize(pumps)
  repeatSize = calculateRepeatSize(pumps)

  if complexity == "polynomial":
    maxRepetitions = min(ceil(pow(remainSteps / repeatSteps, 1 / degree)), floor((maxSize - fixedSize) / repeatSize))
  else if complexity == "exponential":
    maxRepetitions = min(ceil(log(remainSteps / repeatSteps) / log(2)), floor((maxSize - fixedSize) / repeatSize))
  attackPattern = "s".repeat(maxRepetitions)
  return attackPattern
```

We introduced a concise and structured description of the ReDoS attack based on automaton. It emphasizes the class's role in representing a witness for a regular expression attack, including its ability to generate attack strings and patterns based on the provided pump pairs and suffix.

Algorithm 2: ReDoS based on Automaton

Input: a pre-pump or a pump, a suffix
Output: a list where the `n`-th element is an attack string with `n` repetitions

```
begin
  function mapWitness[B](f: A => B): Witness[B]
  newPumps = []
  for (pre, pump) in pumps:
    newPumps.append((pre.map(f), pump.map(f)))

  newSuffix = suffix.map(f)

  return Witness(newPumps, newSuffix)

function buildAttack(n: Int): Seq[A]
  attack = []

  for (pre, pump) in pumps:
    attack.append(pre)
    for i in range(n):
      attack.append(pump)
  attack.append(suffix)
  return attack

function buildAttackPattern(n: Int)(implicit ev: A =:= UChar): AttackPattern
  transformedPumps = []
  for (s, t) in pumps:
    transformedPumps.append((UString.from(s.map(ev)), UString.from(t.map(ev)), 0))

  return AttackPattern(transformedPumps, suffix, n)
```

We introduced the third algorithm to craft a ReDoS attack string based on fuzzing technique. It designed to translate a string with embedded repeating patterns into a structured Attack Pattern. It accomplishes this by iterating through the string, identifying and extracting repeating portions along with their associated repetition counts and any preceding fixed parts. These extracted repeating sections, known as pumps, are then combined with the remaining fixed parts (the suffix) to form the final Attack Pattern. The process of generating attack strings based on the identified repeating structure, simplifying attack construction and enabling developers to focus on the core attack logic rather than the intricacies of manipulating complex strings with repetition.

Algorithm 3: ReDoS based on fuzzing

Input: a repetition count n, a string seq
Output: attack string pattern

```
begin
  pumps <- create_new_sequence()
  str <- create_new_string_builder()

  pos <- 0

  while pos < size_of(seq):
    match seq[pos]:
      case Wrap(u):
```

```
        pos <- pos + 1
        str.append(as_string(u))
    case Repeat(m, size):
        pos <- pos + 1
        repeat <- n + m
        if repeat > 1:
            s <- create_UString(str.result())
            str.clear()
            pump <- map_slice(seq, pos, pos + size, element => as_string(element))
            pump_string <- join(pump)
            t <- create_UString(pump_string)
            add_to_sequence(pumps, (s, t, m))
            pos <- pos + size

  suffix <- str.result()

  return create_AttackPattern(pumpResult, suffix, n)
```

As shown in Figure 6, the algorithm is assumed to be based on automata theory (Checker.Automaton) at first and performs NFA conversion, then falls back to the fuzzing (Checker.Fuzz) if the size of the NFA exceeds the threshold.

Algorithm 4: ReDoS detection based on Automata theory and Fuzzing

Input: a source, flags, a pattern and params
Output: Result of Detection's algorithm
begin

```
  maxNFASize = if params.checker == Checker.Auto then params.maxNFASize else Integer.MAX_VALUE

  result = Try():
    if params.checker == Checker.Auto and repeatCount(pattern) >= params.maxRepeatCount:
        Checker.Fuzz
    else:
        Success(())

    complexity = if pattern.isConstant:
        Success(Iterator.empty)
    else:
        if params.checker == Checker.Auto and pattern.size >= params.maxPatternSize:
            return Checker.Fuzz
        else:
            Success(())
        epsNFA = EpsNFABuilder.build(pattern)
        orderedNFA = epsNFA.toOrderedNFA(maxNFASize).rename().mapAlphabet(lambda x: x.head)
        return AutomatonChecker.check(orderedNFA, maxNFASize)

  for cs in result:
    for vul in cs:
        if vul is Vulnerable:
            attack=vul.buildAttackPattern(params.recallLimit, params.maxRecallStringSize)
            return Diagnostics.Vulnerable(source, flags, vul.toAttackComplexity(), attack, vul.hotspot, Checker.Automaton)
        else if vul is Safe:
            return Diagnostics.Safe(source, flags, vul.toAttackComplexity(), Checker.Automaton)
    ) ++ Iterator(Diagnostics.Safe(source, flags, AttackComplexity.Safe(false), Checker.Automaton))
    )
```

IV. EXPERIMENT

TABLE 2: REGEX SETS FOR EVALUATION

Name	Number	Description
RegLib [19]	2,992	Online regex examples from RegExLib.com
Snort [20]	12,499	Regexes extracted from the Snort NIDS for data packet filtering
Corpus [20]	13,597	Regexes from scraped Python projects

This research presents a comprehensive evaluation of different techniques for detecting Regular Expression Denial of Service (ReDoS) vulnerabilities. The experiment utilizes a large dataset of 29,088 regexes collected from Table 1, serving as a benchmark for testing the effectiveness of various methods. Five techniques are compared: Our technique, ReScue [14], SlowFuzz [14], RXXR2 [16] and Rexploiter [17], and NFAA [18].

Each technique is evaluated by running it on the dataset and assessing the success rate of generating ReDoS strings.

All experiments are conducted on a high-performance server with 24 cores Intel Xeon Gold 6226R, 64 GB RAM DDR4 ECC 2933 MHz and Ubuntu 22.04 LTS. The study's comprehensive approach and detailed analysis provide valuable insights into the strengths and weaknesses of different ReDoS detection methods, aiding researchers and developers in choosing the most appropriate tool for their needs.

We selected 227 samples containing ReDoS vulnerabilities and conducted 10 experiments on this chosen dataset. The results show that ReSCUE and our technique exhibit the highest number of identified vulnerable regexes, indicating their effectiveness in detecting potential vulnerabilities. However, the lack of TP rate for these methods makes it difficult to directly compare their accuracy. SlowFuzz achieves a lower detection rate, has a considerably higher TP rate (31.3%), suggesting better precision. RXXR2 boasts a high TP rate (67.8%) but suffers from a large number of false positives, making it less reliable. Rexploiter relatively low TP rate (1.9%), shows the highest number of false positives, highlighting the potential for inaccurate detection. NFAA does not detect any vulnerabilities, indicating its inadequacy in handling this type of vulnerability. The experiments emphasizes the importance of our technique research to develop more accurate and reliable methods for detecting and attacking regex vulnerabilities.

TABLE 3: THE OVERALL EVALUATION RESULTS

Technique	**Number Vul**	**Number FP**	**TP Rate**	**Avg Time (s)**
New Technique	155 (68%)	-	-	0.6523
ReSCUE	179 (79%)	-	-	0.7154
SlowFuzz	99 (44%)	45	31.3%	0.5351
RXXR2	120 (53%)	57	68%	0.0034
Rexploiter	40 (18.6%)	2084		0.3512
NFAA	0 (0%)	712	N/A	2.5122
Summary 227 (100%)				

V. CONCLUSION

This research presents a novel approach to detect ReDoS vulnerabilities by combining the strengths of both automata-based algorithms and fuzzing techniques. We combined dynamic and static analysis to detect and generate attack strings for ReDoS vulnerabilities. This hybrid approach aims to address the limitations of individual methods, achieves a balance between detection speed, accuracy, and attack string generation effective.

The evaluation results demonstrate the effectiveness of the proposed approach, detect a significant number of vulnerabilities with a reasonable accuracy. The implementation can effectively identify ReDoS vulnerabilities in a variety of scenarios, offer a valuable tool for security researchers and developers. In future research, we will focus on enhancing the detection of ReDoS vulnerabilities based on the techniques of automaton and fuzzing, and subsequently generating more effective attack strings. Overall, this research provides a promising solution for effectively detecting ReDoS vulnerabilities and contributes significantly to the field of regular expression security.

REFERENCES

[1] OWASP (2010-02-10). "Regex Denial of Service". Retrieved 2010-04-16.

[2] Son, D. T., Tram, N. T. K., & Thu, T. T. . (2022). Machine learning approach detects DDoS attacks. Journal of Science and Technology on Information Security, 1(15), 102-108. https://doi.org/10.54654/isj.v1i15.850.

[3] Martin Berglund, Frank Drewes, Brink van der Merwe. 2014. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. Electronic Proceedings in Theoretical Computer Science 151(Proc. AFL 2014).

[4] Efe Barla, Xin Du, James C. Davis. 2023. Exploiting Input Sanitization for Regex Denial of Service. Proceedings of the ACM/IEEE 44th International Conference on Software Engineering (ICSE) 2022. https://arxiv.org/abs/2303.01996.

[5] "Backtracking in .NET regular expressions - .NET". learn.microsoft.com. 11 August 2023. When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to RegularExpressions, causing a Denial-of-Service attack. ASP.NET Core framework APIs that use RegularExpressions pass a timeout.

[6] Li, Yeting, et al. "*ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection.*" 30th USENIX Security Symposium (USENIX Security 21). 2021.

[7] Davis, James C., Francisco Servant, and Dongyoon Lee. 2021. "*Using selective memoization to defeat regular expression denial of service (ReDoS).*" 2021 IEEE Symposium on Security and Privacy (SP), Los Alamitos, CA, USA.

[8] Paul Wilton. Beginning JavaScript. John Wiley & Sons, 2004.

[9] Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with BEK. In USENIX Security Symposium, pages 1–16, August 2011.

[10] https://en.wikipedia.org/wiki/Thompson's_construction.

[11] Sugiyama, Satoshi, and Yasuhiko Minamide. "Checking time linearity of regular expression matching based on backtracking." Information and Media Technologies 9.3 (2014): 222-232.

[12] Shen, Yuju, et al. "ReScue: crafting regular expression DoS attacks." 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2018.

[13] Chida, Nariyoshi, and Tachio Terauchi. 2020."*Automatic repair of vulnerable regular expressions.*" arXiv preprint arXiv:2010.12450.

[14] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In Proceedings of the International Conference on Computer and Communications Security (CCS ’17). 2155–2168. https://doi.org/10.1145/3133956. 3134073.

[15] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static analysis for regular expression denial-of-service attacks. In Proceedings of the 7th International Conference on Network and System Security (NSS ’13). 135–148. https://doi.org/ 10.1007/978-3-642-38631-2_11.

[16] Valentin Wüstholz, Oswaldo Olivo, Marijn JH Heule, and Isil Dillig. 2017. Static detection of DoS vulnerabilities in programs that use regular expressions. In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’17). 3–20. https://doi.org/10.1007/ 978-3-662-54580-5_1.

[17] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce Watson. 2016. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In Proceedings of the International Conference on Implementation and Application of Automata (CIAA ’16). 322–334. https: //doi.org/10.1007/978-3-319-40946-7_27.

[18] Weber, Andreas, and Helmut Seidl. 1991. "*On the degree of ambiguity of finite automata.*" Theoretical Computer Science 88.2 (1991): 325-349.

[19] Asiri Rathnayake and Hayo Thielecke. 2014. Static analysis for regular expression exponential runtime via substructural logics. (2014). arXiv:arXiv:1405.7058.

[20] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA ’16). 282–293. https://doi.org/10.1145/ 2931037.2931073.

ABOUT THE AUTHOR

Nguyen Trung Dung

Workplace: Research Institute 486, Command 86

Email: ntdtoanud2011@gmail.com

Education: 1996-2002: Engineer of Information Security; 2006-2008: Master of Information Security; 2011-2019: Doctor of Philosophy.

Recent research direction: information security, information technology.

Tên tác giả: **Nguyễn Trung Dũng**

Cơ quan công tác: Viện nghiên cứu 486, Bộ Tư lệnh 86

Email: ntdtoanud2011@gmail.com

Quá trình đào tạo: 1996-2002: Kỹ sư chuyên ngành ATTT; 2006-2008: Thạc sỹ chuyên ngành An toàn thông tin; 2011-2019: Tiến sỹ.

Hướng nghiên cứu hiện nay: An toàn thông tin, công nghệ thông tin, bảo mật thông tin.

Pham Van Toi

Workplace: Research Institute 486, Command 86

Email: toiphamvp@gmail.com

Education: 2009-2015: Engineer of automation systems; 2015-2019: Doctor of Philosophy

Recent research direction: information security, information technology.

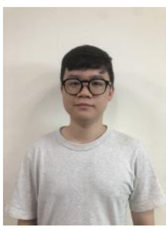
Tên tác giả: **Phạm Văn Tới**

Cơ quan công tác: Viện nghiên cứu 486, Bộ Tư lệnh 86

Email: toiphamvp@gmail.com

Quá trình đào tạo: 2009-2015: Kỹ sư chuyên ngành các hệ thống tự động hóa; 2015-2019: Tiến sỹ

Hướng nghiên cứu hiện nay: An toàn thông tin, công nghệ thông tin, bảo mật thông tin.



Phung Minh Hieu

Workplace: Research Institute 486, Command 86

Email: pmhieu22@gmail.com

Education: 2015-2020: Engineer of Information Security; 2020-2022: Master of Information Security

Recent research direction: information security, information technology.

Tên tác giả: **Phùng Minh Hiếu**

Cơ quan công tác: Viện nghiên cứu 486, Bộ Tư lệnh 86

Email: pmhieu22@gmail.com

Quá trình đào tạo: 2015-2020: Kỹ sư An toàn thông tin; 2020-2022: Thạc sĩ An toàn thông tin

Hướng nghiên cứu hiện nay: An toàn thông tin, công nghệ thông tin.