

國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

利用分散式正規表式比對保護 URL 檢查之隱私

Distributed Regular Expression Matching for
Privacy-Preserving URL Checking

賴侃軒

Kan-Hsuan Lai

指導教授: 蕭旭君 博士

Advisor: Hsu-Chun Hsiao, Ph.D.

中華民國 112 年 8 月

August, 2023

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

利用分散式正規表式比對保護 URL 檢查之隱私

Distributed Regular Expression Matching for Privacy-
Preserving URL Checking

本論文係賴侃軒君（學號 R10922078）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 112 年 7 月 24 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 24 July 2023 have examined a Master's thesis entitled above presented by LAI, KAN HSUAN (student ID: R10922078) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

蕭旭君

(指導教授 Advisor)

陳昱沂

游家敏

系主任/所長 Director:

洪士瀨



誌謝

感謝我的指導教授給予我許多的協助讓我順利完成碩士論文。



摘要

近年來，URL 檢查由於安全瀏覽服務的廣泛使用，因此其隱私問題變得愈發重要。有鑑於此，我們提出了一個保護隱私的 URL 檢查系統，其特色為將使用者的 URL 分割成若干份秘密以保護使用者的隱私。我們的系統利用多個計算節點對不同份秘密進行正則表達式匹配，因此只要至少有一個節點被視為半誠實的，則所有節點和伺服器都無法獲取所有的秘密以還原使用者的 URL，從而保護了使用者的隱私。我們用 Python 語言打造了系統的概念證明，並優化了我們的系統，以在 URL 檢查的情境中更高效地進行計算並減少網絡流量的傳輸。我們在 Easylist 廣告域數據集上評估了我們系統的整體性能。實驗結果顯示，我們的優化技術在一個簡易的測試中能將計算時間從 113.10 秒大幅縮短至 0.04 秒，而在一般情況下，優化後的系統完成一個匹配過程只需要約 3 至 4 秒，非常高效。

關鍵字：正規表式比對、隱私保護、URL 檢查、秘密分享、DFA



Abstract

Recently, as URL checking becomes more and more popular because of widely used Safe Browsing service, its privacy issue becomes more and more important. Therefore, we propose a privacy-preserving URL checking system which splits the user's URL into secret shares to protect the user's privacy. Our system utilizes several computing nodes to compute regular expression matching result on different secret shares, so that as long as at least one node is considered semi-honest, none of the nodes and the server could obtain all the secret shares to reveal the user's URL, and thus the user's privacy is preserved. We design a proof-of-concept of our system in Python, and optimize our system to compute more efficiently and transmit less network traffic in URL checking application. We evaluate our system's overall performance on Easylist ad domain dataset. The result shows that our optimization techniques greatly accelerates the matching process from 113.10s to 0.04s in a simple case, and our well-optimized system takes about 3 ~4s to finish a matching process in average case, which is quite efficient.

Keywords: regex matching, privacy-preserving, URL checking, secret sharing, DFA



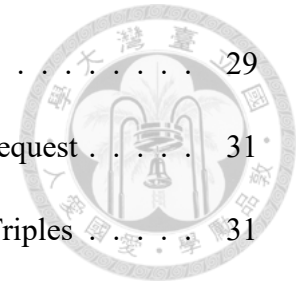
Contents

	Page
口試委員會審定書	i
誌謝	ii
摘要	iii
Abstract	iv
Contents	v
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Background	5
2.1 Notations	5
2.2 Deterministic Finite Automaton	7
2.3 Regex Transformation	8
2.4 Secret Sharing	9
2.4.1 Split a Secret	9
2.4.2 Reconstruct a Secret	10
2.5 Computations on Secret Sharing Domain	10
2.5.1 XOR Operation on the Secret Sharing Domain	11



2.5.2	NOT Operation on the Secret Sharing Domain	11
2.5.3	AND Operation on the Secret Sharing Domain	12
2.5.4	OR Operation on the Secret Sharing Domain	14
3	Problem Definition	15
3.1	Entities	15
3.2	Threat Model	16
3.3	Design Goal	17
4	Proposed Method	18
4.1	Overview	19
4.2	System Architecture	20
4.2.1	User	20
4.2.2	Server	21
4.2.3	Computing Node	21
4.3	Regex Matching on the Secret Sharing Domain	21
4.3.1	Split DFA Matrix into Secret-Shared Form	22
4.3.2	Comparison between Secret Shares on Secret Sharing Domain	23
4.3.3	Obtain Value from Secret-Shared DFA Matrix	24
4.3.4	Evaluate Result on Secret-Shared DFA Matrix	26
5	Implementation	27
5.1	Basic Implementation	27
5.1.1	Generate Secret-Shared DFA Matrix	27
5.1.2	Setup	28
5.1.3	Evaluate Result	28

5.2	Optimization 1: Compare Input and State Separately	29
5.3	Optimization 2: Perform Independent-ANDs in One Request	31
5.4	Optimization 3: Use PRNG to Generate Split Beaver Triples	31
5.5	Optimization 4: Reduce Input Character Set	32
6	Evaluation	33
6.1	Dataset	33
6.2	Complexity Analysis	34
6.2.1	Complexity of DFA Size	34
6.2.2	the Number of AND Operations	37
6.2.3	Beaver Triple Length	38
6.3	Security Analysis	39
6.4	Computation Time and Network Traffic Size	41
7	Related Work	46
8	Discussion and Future Work	48
8.1	Verifiability	48
8.2	Availability	49
8.3	Evaluation on Multiple DFAs	49
8.4	Limitation	50
9	Conclusion	51
	Bibliography	52





List of Figures

2.1	Plaintext domain and share secret domain	6
2.2	An example of splitting data	10
2.3	An example of reconstructing result	10
3.1	The relationship between user and server. The user query the server to check the URL while avoiding the server from knowing the URL; the server evaluate and return the query result to the user without leaking information of the dataset.	15
4.1	The procedure of privacy-preserving URL checking algorithm	19
4.2	System architecture	20
4.3	An example of how to obtain value from an array given the index	24
6.1	DFA sizes of easylist filters	36
6.2	DFA sizes of easylist filters excluding extreme values	36
6.3	DFA sizes of easylist filters excluding extreme values and $\frac{\Delta y}{\Delta x} > 2$. The orange line is the cubic regression result: $y = 1.14x + 1.84 \times 10^{-2}x^2 + 4.83 \times 10^{-5}x^3$	37
6.4	Distribution of DFA size: 98.98% of URL patterns (43021) with less than 100 states	43



List of Tables

6.1	Results of evaluating input = "agg" and regex = ".*g"	41
6.2	Average result of evaluating input length = 66	43
6.3	Result of evaluating DFA size = 54	44

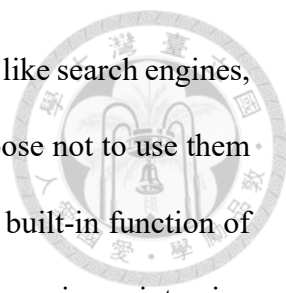


Chapter 1

Introduction

With the exponential growth of the Internet and its pervasive influence on modern society, web browsing has become an integral part of our daily routine. Whether for work, entertainment, or information gathering, we rely heavily on web browsers to explore the vast virtual landscape. However, the convenience and accessibility offered by web browsing come hand in hand with a range of security threats that can compromise our digital well-being. There are a lot of malicious websites on the Internet that may contain fake information, download malware for users, phishing, and so on. To prevent users from accidentally access the malicious websites, there are a lot of URL checking services that help users to filter the malicious URLs and make web-browsing safer.

While constantly checking for URL safeness protects the users from cyber threats, their online privacy may be affected because all the browsing histories are recorded by the URL checking service. The privacy issue becomes more severe with a type of service called safe browsing, which is adopted by most of the major browsers, including Chrome, Safari and Firefox. A browser with the safe browsing feature will check all the URLs before fetching, which works automatically in the background, so that the user may be unaware



of it. There are other kinds of services that may have trackers, though, like search engines, social media networks, and e-commerce platforms, but users can choose not to use them if they distrust the companies; safe browsing, on the other hand, is a built-in function of browsers and essential for users to browse safely, which makes it more privacy intrusive than other types of trackers. Besides, because it is built in the browser, the user base of safe browsing is large, making it more motivating of being used as a tracker. For example, Google Safe Browsing serves about 5 billion devices, and detected about 3 to 5 million threats per week in 2022 [13].

To ease the privacy issue, Google Safe Browsing provides an alternative API for users to check their URLs privately. Google Safe Browsing provides two APIs, Lookup and Update, and the latter is the privacy-preserved version of the former. The Lookup API matches the plaintext URL, while the Update API matches the URL's hash prefix, thus preserving the user's privacy. However, the Update API also has two drawbacks: first, because of hash prefix, one can only do exact comparisons during URL checking, which lacks flexibility; second, it is pointed out [9] that even the Update API sometimes leaks information of user's URL and endangers the user's privacy due to canonicalization. The leak occurs because hash prefix can only be used to perform exact matching, so the user has to generate different decompositions and search for all their hash prefixes to eliminate false negatives, where the decompositions are the combinations of all subdomains and subpaths of the target URL [9]. The problem is when the user queries two or more hash prefixes of decompositions, the server has more information to guess what URL the user is actually browsing. What makes it worse is that if the safe browsing service provider wants to track a specific website, the provider can inject the hash prefix of that website into the user's local database (while the target website is safe) to force the user to query

such a hash prefix when browsing the target website.

In this paper, we aim for a solution of privacy-preserving URL checking that ensures no information of user's URL is leaked and supports flexibility when checking URLs. More specifically, we aim to support regular expression (regex) matching so that the safe browsing service providers can design malicious URL patterns with less limitation, while protect user's URLs from being known by others, including the service provider. In addition to the above goals, we also aim to outsource the workload from the user as much as possible since we expect that the user's device may have little computing resource, such as portable devices or IoT devices. To achieve the requirements, we introduced **regex matching on shared secret**: the user's URL is split into secret shares, and as long as no one has enough number of shares, anyone except the user cannot reconstruct any secrets including the user's URL and the matching result, and thus the user's privacy is preserved.

In our system, we introduce some third-party computing nodes to perform matching on different secret shares. The most challenging problem is how to efficiently perform regular expression (regex) matching on secret shared data. We adopt N -out-of- N XOR-based secret sharing, where AND and NOT have been implemented, and thus it is functionally-complete. However, when computing AND, it requires a synchronization between nodes and an additional pre-generated message, called **beaver triple**, which is both time consuming and network bandwidth consuming for data transmission. Therefore, we have to design the system that uses as few ANDs as possible. We evaluate our well optimized system on Easylist ad domain dataset [8], and the result shows that our system takes about 3.7 second to evaluate the matching result of average length pattern (54 states) and URL (66 characters), which is quite efficient.

Our contributions are listed as follows:

- We solve the privacy issue of URL checking by utilizing secret sharing.
- We support regex matching to expand the application scenarios.
- We design several optimization techniques to minimize the synchronization time and network traffic.



In Chapter 2, we will introduce some background knowledge needed for our solution; in Chapter 3, we will define our problem and the threat model of our problem; in Chapter 4, we will explain how to perform regex matching on share secret; in Chapter 5, we will introduce the system of our solution and some optimization techniques to make the system more efficient; in Chapter 6, we will analyze the complexity of our system and overall performance with an ad domain dataset; in Chapter 7, we will introduce some related works; in Chapter 8, we will discuss the limitation of our system and some possible future work.



Chapter 2

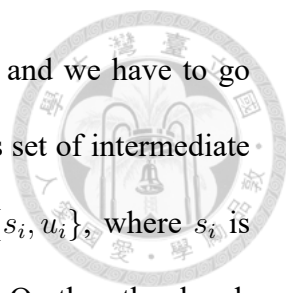
Background

In this chapter, we will introduce the notations we use, why and how we transform regex to DFA, and how we compute on secret shares.

2.1 Notations

In the following context, all the variables with square brackets are secret shares of the original message. For example, for arbitrary message x , $[x]_n$ is denoted as the n th secret share of x , and is used by n th computing node. $\{[x]_n\} \triangleq \{[x]_i \mid 1 \leq i \leq N\}$ is denoted as the set of all secret shares of x . $Split(x)$ means to generate secret shares from the original message, whereas $Reconstruct(\{[x]_n\})$ means to reconstruct the original message from secret shares. $x \oplus y$ is the logical XOR operation, x' is the logical NOT operation, $x + y$ is the logical OR operation, and $x \cdot y$ or xy are the logical AND operation. \parallel represents for string concatenation.

We say that x is on the **plaintext domain** and $[x]_n$ is on the **secret sharing domain**. The relationship between plaintext domain and secret sharing domain is shown in Figure 2.1.



Suppose that V_0 is the parameter set of a URL checking algorithm, and we have to go through a series of functions (f_1, f_2, \dots) to get the final result r . V_i is set of intermediate values of the algorithm, which, in the case of regex matching, is $\{s_i, u_i\}$, where s_i is the intermediate state of DFA and u_i is the i th character of the URL. On the other hand, $r \in \{0, 1\}$ is the matching result, where 0 means **not match** and 1 means **match**. f_i are functions to obtain next state by $f_i(s_i, u_i) = DFA[s_i, u_i]$. We say that (f_1, f_2, \dots) are functions on the plaintext domain because both their parameters and results are in plaintext form.

For each function f_i , we aim to design an equivalent function f'_i that its parameters and result are in the secret sharing form; that is, $f_i(V_{i-1}) = Reconstruct(f'_i(Split(V_{i-1})))$.

We say that (f'_1, f'_2, \dots) are functions on the secret sharing domain.

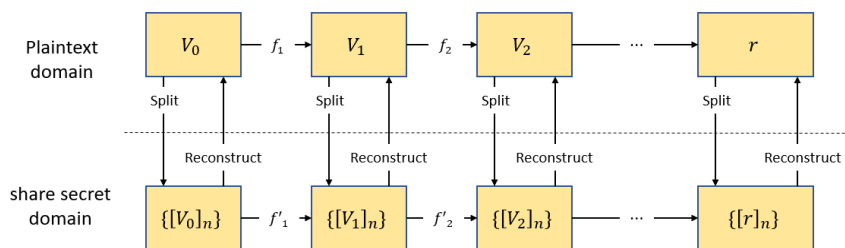


Figure 2.1: Plaintext domain and share secret domain

If we want to evaluate a function $r = f(x, y)$ on the secret sharing domain, the process is as follows:

Step 1: $\{[x]_n\} \triangleq Split(x), \{[y]_n\} \triangleq Split(y)$

Step 2: Send $[x]_n, [y]_n$ to the n th node

Step 3: n th node evaluate $[r]_n \triangleq f([x]_n, [y]_n)$

Step 4: $r \triangleq Reconstruct(\{[r]_n\})$

If an operator is written in a function form, such as $AND()$, the function is on the secret sharing domain. For a function on the secret sharing domain, we need to split its parameters beforehand. As for the result of the function, if it is the final result, then we need to reconstruct it to reveal the final result; otherwise, it is an intermediate value of the algorithm and it is used directly as the parameter of the subsequent function.

2.2 Deterministic Finite Automaton

Deterministic Finite Automaton (DFA) is a finite-state machine that determines whether an input is accepted or not [21]. A DFA consists of five parameters: $M = \{Q, \Sigma, \delta, s, F\}$.

Q : The finite set of all states

Σ : The finite set of characters

δ : The transfer function, which takes current state and an character as input, and output the next state

s : The initial state

F : The accepted state set

A finite-state machine has a state and takes a sequence of characters as input. The machine iterates through the input characters, and continuously updates its state according to the transfer function δ . Equivalently, a DFA can be expressed in the matrix form $M(s_t, u_t) = s_{t+1}$. That is, given a current state s_t and an input character u_t , the DFA matrix specifies the next state s_{t+1} . Evaluation on a DFA matrix is simple: simply iterate through the input

string and update the current state until reaching the end, then determine whether the final state is in the accepted state F or not.



2.3 Regex Transformation

A regex needs to be compiled into NFA or DFA before matching. Typically, evaluating on NFA consumes less memory and supports more features such as back-referencing, yet its time complexity is higher than evaluating on DFA and the matching process on NFA is also more complex than on DFA. In our system, we choose to transform regex to DFA because we have to implement the matching process by logic circuits, so it is vital to keep the matching process simple and fast.

The transformation between regex and DFA has been studied over 50 years, and many algorithms have been developed with varying complexities [15]. Roughly speaking, there are three steps to transform a regex to a DFA: regex to Non-Deterministic Finite Automaton (NFA) conversion, NFA to DFA conversion and DFA minimization. Below are some algorithms that can be used in each step:

regex to NFA: Thompson's construction [25], Glushkov's construction [10]

NFA to DFA: Powerset construction [22]

DFA minimization: Hopcroft's algorithm [16], Moore's algorithm [19]

Beside the three-step transformation, there are also other algorithms that transform regex directly to DFA, such as partial derivative automaton. Since it has been developed for many years, the regex engines have become optimized and efficient. We can utilize well-developed regex engines, extracting the DFA matrix for our use.



2.4 Secret Sharing

Secret sharing is a cryptographic technique that involves dividing a secret into multiple shares and distributing them among participants. Each share contains partial information about the secret. By combining a required number of shares, the original secret can be reconstructed. This approach ensures that no single participant possesses the entire secret, enhancing security. Secret sharing has applications in various fields, including cryptography, secure multi-party computation, and key management systems, providing a robust method for safeguarding sensitive information.

In the following context, we assume that there are only two computing nodes in the system and we only generate two shares from each secret. However, our system also works when there are more than two nodes.

2.4.1 Split a Secret

There are multiple ways to split a secret. However, we need a special way to split them, so that we can do desired computations on the secret sharing domain. Among all the options, XOR is simple to calculate and suitable for logical operations, so we choose XOR to split the secret shares. To split an arbitrary secret U , we generate a random byte string x which is as long as U , and then split U into $[U]_1, [U]_2$ by $[U]_1 = x$ and $[U]_2 = U \oplus x$. Figure 2.2 shows an example of splitting data $0xb7$: the first share $0x6f$ is randomly generated and the second share $0xd8$ is derived from the plaintext data and the first share.

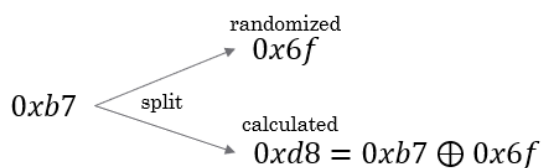


Figure 2.2: An example of splitting data

2.4.2 Reconstruct a Secret

To reconstruct the original message, we simply need to XOR all the secret shares. For the above example, if we want to reconstruct U , we can calculate $U = [U]_1 \oplus [U]_2$. Figure 2.3 shows an example of reconstructing the result from two shares, $0x01$ and $0x00$.

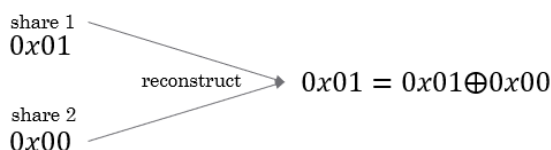
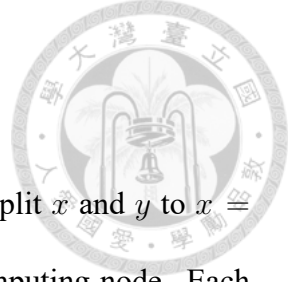


Figure 2.3: An example of reconstructing result

2.5 Computations on Secret Sharing Domain

We can compute any function on secret sharing domain by supporting a functionally complete set [20]. First, we will show that we can compute XOR and NOT on share secret domain using the characteristic of XOR. Then, we will show that by introducing **beaver triples** (a, b, c) , we can also compute AND and OR. Finally, because AND, NOT is a functionally complete set, we are able to compute any function on secret sharing domain.



2.5.1 XOR Operation on the Secret Sharing Domain

Suppose we want to calculate $r = XOR(x, y)$. To do so, first we split x and y to $x = [x]_1 \oplus [x]_2$ and $y = [y]_1 \oplus [y]_2$. Next, send $[x]_n, [y]_n$ to the n th computing node. Each computing node then calculates $[r]_n = [x]_n \oplus [y]_n$ and sends $[r]_n$ back to the user. The result is correct because

$$r = [r]_1 \oplus [r]_2 = [x]_1 \oplus [y]_1 \oplus [x]_2 \oplus [y]_2 = x \oplus y$$

Note that from the node's perspective, calculating XOR on the plaintext domain or on the secret sharing domain makes no difference, so the notation is also the same.

2.5.2 NOT Operation on the Secret Sharing Domain

The implementation of NOT is relatively simple. We can calculate $r = x'$ by using the fact that

$$(a \oplus b)' = a \oplus b \oplus 1 = (a \oplus 1) \oplus b = a' \oplus b$$

Therefore, we only need to do NOT on one of the secret shares. Let $[r]_1 = [x]_1'$ and $[r]_2 = [x]_2$; the result will be $r = [x]_1' \oplus [x]_2 = x'$. To determine which node should negate its secret share, each node is assigned with a unique index at the beginning, and only the node with the smallest index should negate its secret share. The indexes can be assigned in any order by the user, the server, or even the consensus of all nodes, because the node with the smallest index doesn't have any additional information than other nodes.

We will use $NOT(x)$ to denote x' on the secret sharing domain in the following context.



2.5.3 AND Operation on the Secret Sharing Domain

The implementation of AND is relatively complex and not intuitive. We need additional information to help calculating it, which is called **Beaver triple** [2]. The Beaver triple we use is a variant of the original version — the original one works on arithmetic circuits, while ours works on logic circuits. A Beaver triple consists of 3 variables a , b and c such that $c = a \cdot b$. To calculate $r = x \cdot y$, first we generate the share secrets of x , y , a , b and c , and send them to their respective computing nodes. The computing process is:

Step 1: each node computes $[x]'_n = [x]_n \oplus [a]_n$, $[y]'_n = [y]_n \oplus [b]_n$

Step 2: The nodes share $[x]'_n$, $[y]'_n$ with each other. All nodes then reconstruct x' and y' .

Step 3: The first node computes

$$[r]_1 = x'y' \oplus [b]_1x' \oplus [a]_1y' \oplus [c]_1$$

The rest of nodes compute

$$[r]_n = b_nx' \oplus [a]_ny' \oplus [c]_n$$

Next, We will show that the result is correct. The proof utilizes the distribution law between logical AND and logical XOR, which can be proven by Truth Table. The result is as follows:



$$\begin{aligned}
r &= \bigoplus [r]_n \\
&= x' y' \oplus \bigoplus ([b]_n x' \oplus [a]_n y' \oplus [c]_n) \\
&= x' y' \oplus \bigoplus [b]_n x' \oplus \bigoplus [a]_n y' \oplus \bigoplus [c]_n \\
&= x' y' \oplus x' \bigoplus [b]_n \oplus y' \bigoplus [a]_n \oplus c \tag{2.1} \\
&= x' y' \oplus b x' \oplus a y' \oplus ab \\
&= (x' \oplus a)(y' \oplus b) \\
&= x \cdot y
\end{aligned}$$

We will use $AND(x, y)$ to denote $x \cdot y$ on the share secret domain in the following context.

AND requires reconstructing $[x]'_n, [y]'_n$ to evaluate the result, which makes it the bottleneck of whole evaluation process because it requires a synchronization between all nodes. However, we can calculate several AND s in parallel as long as they can be calculated concurrently without depending on each other's results. For example, if we want to calculate $x = AND(a, b)$ and $y = AND(c, d)$, then we can combine them to become $x||y == AND(a||c, b||d)$. This way, we can calculate multiple AND s in one synchronization process and thus the synchronization time is reduced. A counterexample is that if we want to calculate $r = x \cdot y \cdot z$ on the share secret domain, we have to calculate AND 2 times, which is $r = AND(AND(a, b), c)$, because we have no 3-input AND . In this situation, we cannot compute two AND s in parallel because the second AND requires the result of first AND .

2.5.4 OR Operation on the Secret Sharing Domain



To implement OR, we can utilize De Morgan's Law

$$a + b = (a' \cdot b)'$$

Therefore, we can combine AND and NOT to implement OR. Notice that although OR requires 3 more NOT operations, it does not consume more time than AND to compute because NOT is a constant time function, which is much faster than AND.

We will use $OR(x, y)$ to denote $x + y$ on the secret sharing domain in the following context.



Chapter 3

Problem Definition

In this chapter, we will introduce entities in our problem setting, what characteristics they should have, and the threat model.

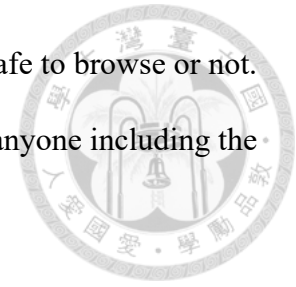
3.1 Entities

There are two entities in the system: a *user* and a *server*. Figure 3.1 shows the relationship between the user and the server.



Figure 3.1: The relationship between user and server. The user query the server to check the URL while avoiding the server from knowing the URL; the server evaluate and return the query result to the user without leaking information of the dataset.

A *user* is the one that holds a URL and want to check whether it is safe to browse or not. Due to the privacy concern, he is not willing to share the URL with anyone including the server.



A *server* is a service provider that holds a dataset of malicious URL patterns. It accepts queries from the user and returns whether the user's URL matches any patterns in the set. The service provider may invest time and money to analyze and design the patterns, so it is also not willing to share the dataset with anyone to protect the intellectual property. We assume that the patterns are written in regex because regex is very powerful and can cover most of the use cases.

3.2 Threat Model

The threat model of the system is given as follows:

- The server is considered semi-honest. It matches the user's encrypted URL with its dataset and returns correct results to the user, not forging a fake one, yet it is curious about what the user's URL is.
- The server and user may require some third-party computing nodes to help evaluating the result. The third-party computing nodes will also evaluate correct results, not forging a fake one, while they have a desire to collect private information from the user and the server, including the user's URL and the server's dataset.
- All the packets should not be eavesdropped or tampered with by any middle-person. This can be achieved by applying a secure channel, but is out of scope of this work.
- The availability problem, such as Denial of Service attack, is not considered in our

proposed system. The discussion of this issue is in Chapter 8.



3.3 Design Goal

Below, we list the goals and features that our system should achieve:

Privacy-preserving: Our system should protect both the user's and the server's privacy, including the user's URL, the server's pattern and matching result.

Regex matching: Our system should support regex matching to give pattern designers more flexibility and versatility in their applications.

Low workload on user side: We expect the user's device may have low computing performance due to hardware limitation (portable devices, IoT, etc.), so it is vital to lower the workload of the user as much as possible.

Efficiency: In addition to achieving above goals, our system should compute fast and minimize the use of network traffic.

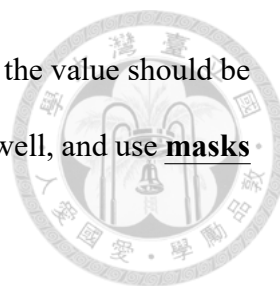


Chapter 4

Proposed Method

In this chapter, we will explain how to determine whether the URL from the user matches the pattern from the server while preserving the user and the server's privacy. Our solution is to utilize N -out-of- N secret sharing to protect the data privacy. In our proposed system, N third-party computing nodes are introduced to help evaluating the result. Notice that if all the nodes collude together, they will obtain all the secret shares and therefore the secret data is leaked. To prevent such attack, we assume at least one node does not join the collusion so that the private data remains safe. The larger N we choose, the more possible that such honest node exists. This idea comes from the Tor Network [5]: the Tor Network protects user's privacy by a series of onion routers, and at least one onion router has to be honest and not collude with each other, otherwise an adversary could reveal the routing path and find out who the user is communicating with.

The main problem of evaluation on secret-shared DFA matrix is how to obtain value from the array when the index is on the secret sharing domain. More specifically, since each node only obtain a share of current-state s_t and input character u_{t+1} to protect privacy, the traditional way to access an array — treating the index as the offset of memory address



— does not work in this situation, so it's hard for a node to tell which the value should be given an secret-shared index. The solution is to split array index as well, and use masks to select the correct value.

In the following part, we will go through the overview of the proposed algorithm and the system architecture, define the notations used in the following context, and introduce our algorithm of performing regular expression matching on secret shares.

4.1 Overview

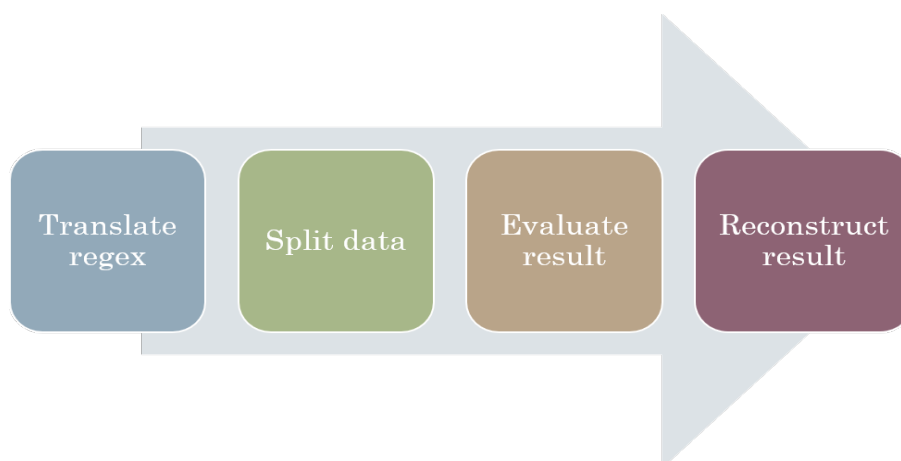


Figure 4.1: The procedure of privacy-preserving URL checking algorithm

Figure 4.1 shows the overview our proposed algorithm. Below is a brief introduction to each step:

Step 1: Translate regex: We assume that all URL patterns are in the regex form. In this step, we will transform the regex pattern into a DFA matrix for the subsequent matching process.

Step 2: Split data: To protect the privacy of the server and the user, we will split DFA matrix and URL in to secret shares before evaluation.



Step 3: Evaluate result: Perform regex matching based on secret-shared DFA and URL.

Step 4: Reconstruct result: reveal the final result to determine whether the URL matches the pattern or not.

4.2 System Architecture

The system includes three types of components: User, Server and Computing Node. Figure 4.2 depicts the relations between the components.

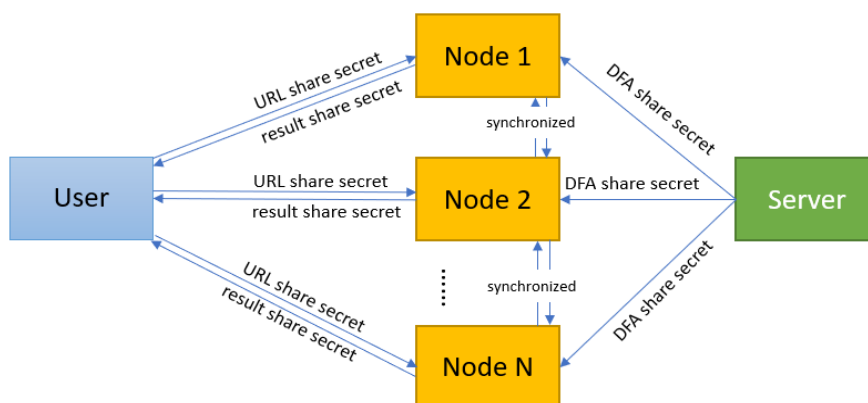


Figure 4.2: System architecture

The functions of each component are shown in the following part.

4.2.1 User

The user holds a URL that he wants to query the server, which is called a **query URL**. To preserve privacy, the user does not want anyone else including the server knows the URL. Instead of sending the URL in plaintext, the user splits the URL into share secrets, sends them to the computing nodes, receive the query result in share secret form from the computing nodes, and finally reconstruct the result.



4.2.2 Server

The server holds a database of URL blacklist. The data in the database is in the form of regular expression string (regex)—that is, if a URL matches the regex, it is labeled as a malicious website. To determine whether the user’s query URL matches a regex, the server first transforms the regex into a Deterministic Finite Automaton (DFA) matrix M , splits M into share secrets, and then sends them to the computing nodes for evaluation. Notice that the server cannot get the secret shares of the query URL, or it can reconstruct the query URL and therefore violate the privacy requirement.

4.2.3 Computing Node

The computing node receives one of the secret shares of query URL from the user, one of the secret shares of M from the server, evaluates the result, and sends the result to the user. Here, every computation is done on the secret sharing domain; that is, any parameters, intermediate values and result are in share secret form. Therefore, though the computing node handles the whole computation, it knows nothing about the query URL and the regex. There should be at least two computing nodes in the system.

4.3 Regex Matching on the Secret Sharing Domain

To perform regex matching on the secret sharing domain, we need to split the DFA matrix, which is generated from the regex, into a secret-shared DFA matrix. Next, we need to let the nodes be able to evaluate the results using the secret-shared DFA matrix. In the following part, we will explain how to split a DFA matrix, obtain value from secret-shared

DFA matrix, and evaluate result on secret-shared DFA matrix.



4.3.1 Split DFA Matrix into Secret-Shared Form

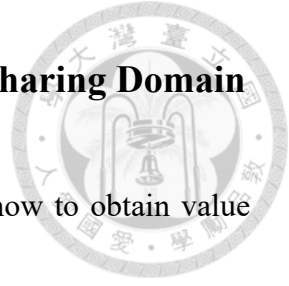
The DFA matrix $M(q, \sigma)$ transformed from regex is 2-dimensional. For simplicity, we first reduce the dimension of M into one dimension. We can simply concatenate the state and input character, and use it as the index (idx) of the 1D DFA array M' . In short, $M'(idx_k) \triangleq M(q_i, \sigma_j)$, where $q_i \in Q$ is the i th state in Q , $\sigma_j \in \Sigma$ is the j th input character in Σ , and $idx_k = q_i || \sigma_j$. We transform (i, j) into the new index k for M' , where $0 \leq k < |Q| \cdot |\Sigma|$ and there exist a one-to-one function f such that $f(i, j) = k$.

To split M' , we have to XOR random bytes with both the index and the value of M' . The secret shares of M' will be:

$$\begin{aligned} [M']_1(k) &= (r_k, R_k) \\ [M']_2(k) &= (idx_k \oplus r_k, M'(idx_k) \oplus R_k) \end{aligned} \tag{4.1}$$

We denote r_k as $[idx_k]_1$ and $(idx_k \oplus r_k)$ as $[idx_k]_2$. r_k and R_k are random byte strings, where r_k is as long as idx_k , and R_k as long as $M'(idx_k)$. Note that when splitting M' , idx_k is no longer unique in a DFA share because uniqueness is not guaranteed when generating random string r_k . Therefore, we change the notation of $[M']_n$ into $[M']_n(k) \triangleq ([idx_k]_n, [M'(k)]_n)$.

4.3.2 Comparison between Secret Shares on Secret Sharing Domain



We should first implement comparison function before discussing how to obtain value from a secret-shared array. More specifically, we want

$$\text{Compare}(x, y) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{if } x \neq y \end{cases}$$

The idea of implementation is using XOR to compare each bit, and then using NOR to merge the compare results of each bit. Only when all the bits of XOR result are 0, will x and y be equal, and thus the comparison result is 1. Suppose x, y are l bits long, then

$$r = \text{XOR}(x, y) \tag{4.2}$$

$$\text{Compare}(x, y) = \text{NOR}(r_0, r_1, \dots, r_l)$$

where r_i means the i th bit of r and l is the length of r .

In practice, we don't have a multi-input NOR operation, so we have to do 2-input NOR $\log(l)$ times, each time merging the result into half length, like the merge sort.



4.3.3 Obtain Value from Secret-Shared DFA Matrix

Suppose $y := M(x)$ and $x = 2$

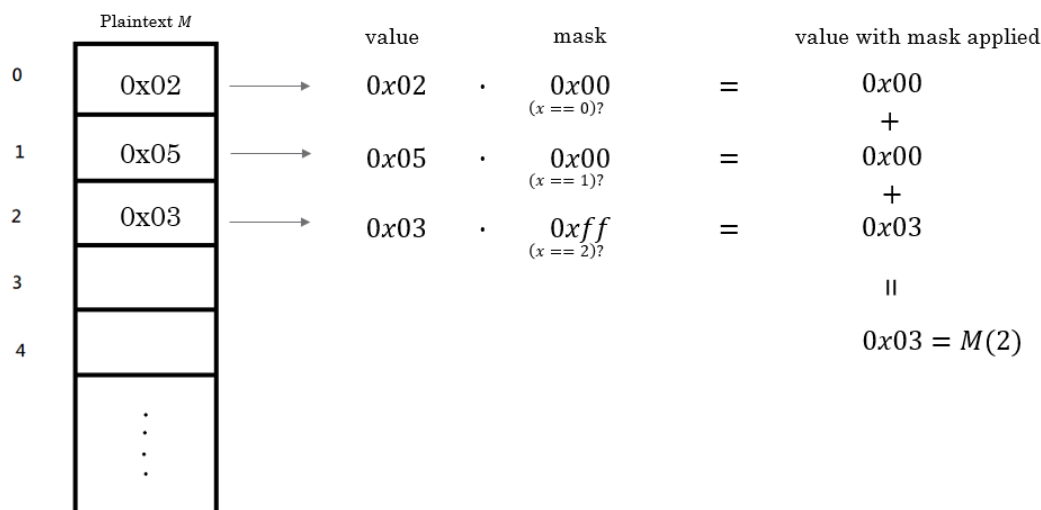


Figure 4.3: An example of how to obtain value from an array given the index

To evaluate results on a secret-shared DFA matrix, we need to know how to obtain values from a secret-shared DFA matrix with a secret-shared index. That is, how to calculate $[M'(x)]_n$ for the n th node with only $[M']_n$ and $[x]_n$. The solution is to use masks to filter the value that the index matches in $[M']_n$. Let us first focus on obtaining value from a plaintext DFA matrix with a plaintext index. Figure 4.3 shows an example of how to obtain value from an array given the index. We don't have a $O(1)$ solution because it is impossible to implement such function on the secret sharing domain, besides it will leak information of what the indexes is to the nodes. Instead, we have to go through all the values and compare the indexes of the values with the given index. We can take the comparison result as a mask, which selects the value that its index matches the given one. In short, we can obtain k from comparing x and idx_k , and set only the k th mask to 1 to select the desired value. Thus, given a plaintext DFA matrix M' and an index x , the function of obtaining value will be:



$$\begin{aligned}
 M'(x) &\triangleq (x == idx_0?) \cdot M'(idx_0) + (x == idx_1?) \cdot M'(idx_1) + \dots \\
 &= \sum_{k=0}^{|\mathcal{Q}| \cdot |\Sigma| - 1} mask_k \cdot M'(idx_k)
 \end{aligned} \tag{4.3}$$

where the length of $mask$ is the same as $M(x)$ and

$$mask_k = \begin{cases} 111\dots 1, & \text{if } x == idx_k \\ 000\dots 0, & \text{else} \end{cases}$$

Note that because only one mask will be 1, there will only be a specific k such that $mask_k \cdot M'(idx_k) \neq 0$. In this special case, the result of XOR is the same as OR. Therefore, we can replace OR with XOR, which generates the same result but brings us an advantage — the implementation of XOR on secret sharing domain doesn't require synchronization, which makes it much faster than OR. Therefore, we modify the formula to become

$$M'(x) \triangleq \bigoplus_{k=0}^{|\mathcal{Q}| \cdot |\Sigma| - 1} mask_k \cdot M'(idx_k)$$

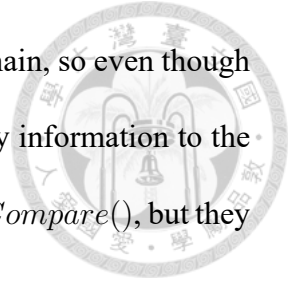
With the above formula, we can now obtain value from the secret-shared DFA matrix by replacing every operation into its secret-shared version. Therefore,

$$[M'(x)]_n = \bigoplus_{k=0}^{|\mathcal{Q}| \cdot |\Sigma| - 1} [AND(mask_k, M'(idx_k))]_n$$

where

$$[mask_k]_n = \begin{cases} 111\dots 1, & \text{if } [Compare(x, idx_k)]_n == 1 \\ 000\dots 0, & \text{else} \end{cases}$$

Note that all the logical operations are done on the secret sharing domain, so even though the *Compare()* function only returns 1-bit result, it does not leak any information to the computing node — each node only knows one of the secret shares of *Compare()*, but they will need all shares to reconstruct the actual result.



4.3.4 Evaluate Result on Secret-Shared DFA Matrix

Finally, we can evaluate the regex matching result with secret shared DFA and URL. By iterating all the input characters, we can obtain the final state and determine whether it is in F or not as the result. Suppose that the length of URL is l , u_i is the i th char of URL U and the initial state is s_0 . The nodes iteratively update the state by $s_{t+1} \triangleq M'(s_t, u_t)$ to obtain the final state s_l . Note that the whole computing process is on the secret sharing domain, and here we just show the equivalent process on the plaintext domain. Finally, they compare s_l with all $f \in F$ where F is the accepted state set of the DFA. If one of the comparison result is 1, then the URL matches the pattern; otherwise, the URL does not match.



Chapter 5

Implementation

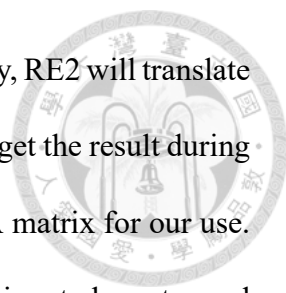
In this chapter, we will introduce the implementation details of the proposed method and three optimization techniques to improve the performance.

5.1 Basic Implementation

We will sequentially introduce each part of the calculation process according to the order of the flow.

5.1.1 Generate Secret-Shared DFA Matrix

The first step is that the server generates the split DFA matrix according to the malicious URL pattern. Here we only focus on translating one pattern each time, and assume that the pattern is in regex form (otherwise translate it to regex first). First, we translate the regex into DFA matrix. We utilize Google RE2[14] engine to do so. RE2 is a regex engine developed by Google, which is widely used in most of Google's products, such as Google Sheet, and is implemented using finite automata while most of the regex engines of major



languages are implemented by recursive backtracking[3]. That is to say, RE2 will translate regex to a DFA matrix, and then trace through the matrix and input to get the result during the matching process. We add a wrapper function to extract that DFA matrix for our use. The extract DFA matrix is an array with two dimensions, which are input character and current state, and the value is the next state. Next, we perform dimension reduction to the 2D DFA matrix by concatenating input character and current state as the index of 1D DFA matrix. Finally, we split both the index and the value of 1D DFA matrix by $Split(DFA)$ to generate split DFA matrixes $[M']_n$, where $n \in [1, N]$ and N is the number of nodes.

5.1.2 Setup

The nodes register themselves to the server to obtain their respective M_n . The user queries the server to obtain the IPs of the nodes, and the length of Beaver triples needed for evaluation process. It then generates a, b of the length given, calculates $c = a \cdot b$, and splits all of them. Finally, it sends the secret-shared Beaver triple $([a]_n, [b]_n, [c]_n)$, and the secret-shared URL to the respective nodes. Note that U is denoted as the URL in the plaintext form, $[U]_n$ is the n th share of U , u_i is the i th character of U , and $[U_i]_n$ is the n th share of the i th character of U .

5.1.3 Evaluate Result

The nodes can evaluate the result with $[M']_n, [a]_n, [b]_n, [c]_n$ and $[U]_n$. Here we take a higher view on the evaluation process — we focus on how to derive the result on the plaintext domain. As mentioned in Chapter 2, we need to evaluate $r = M'(\dots M'(M'(s_0, u_0), u_1)\dots, u_n)$ to get the result. The algorithm is shown in Algorithm 1.



Algorithm 1 evaluate DFA result

Require: M', U, s_0, F

```
1:  $s \leftarrow s_0$ 
2: for  $i \leftarrow 0$  to  $U.length$  do
3:    $ns \leftarrow 0$  // ns = next state
4:   for  $k \leftarrow 0$  to  $|Q| \cdot |\Sigma|$  do
5:     if  $Compare(s||u_i, idx_k) == 1$  then
6:        $mask \leftarrow 111...1$ 
7:     else
8:        $mask \leftarrow 000...0$ 
9:     end if
10:     $ns \leftarrow XOR(ns, AND(mask, M'[idx_k]))$ 
11:  end for
12:   $s \leftarrow ns$ 
13: end for
14:  $DFAResult \leftarrow 0$ 
15: for  $ms \in F$  do // ms = match state
16:    $DFAResult \leftarrow XOR(DFAResult, Compare(s, ms))$ 
17: end for
18: return  $DFAResult$ 
```

5.2 Optimization 1: Compare Input and State Separately

In basic implementation, we squash the DFA matrix into 1D and split the index by simply XOR a random string. To obtain the next state, we need to generate $|Q| \cdot |\Sigma|$ masks. The idea of this optimization is to compare input character and current state separately, and then combine the two masks. Therefore, we only need to generate $|Q| + |\Sigma|$ masks. To do so, the server has to split Σ and Q and sends them to the nodes as additional information. The algorithm is shown in Algorithm 2. Note that here we turn to use the original 2D DFA matrix M instead of 1D version M' . The improvement and discussion will be in Chapter 6.



Algorithm 2 evaluate DFA result with Optimization 1

Require: M, Σ, Q, U, s_0, F

```
1:  $s \leftarrow s_0$ 
2: for  $i \leftarrow 0$  to  $U.length$  do
3:    $ns \leftarrow 0$  // ns = next state
4:    $mask_Q \leftarrow List(length = |Q|)$ 
5:    $mask_\Sigma \leftarrow List(length = |\Sigma|)$ 
6:   for  $\sigma \in \Sigma$  do
7:     if  $Compare(u_i, \sigma) == 1$  then
8:        $mask_\Sigma[\sigma] \leftarrow 111...1$ 
9:     else
10:       $mask_\Sigma[\sigma] \leftarrow 000...0$ 
11:    end if
12:  end for
13:  for  $q \in Q$  do
14:    if  $Compare(s, q) == 1$  then
15:       $mask_Q[q] \leftarrow 111...1$ 
16:    else
17:       $mask_Q[q] \leftarrow 000...0$ 
18:    end if
19:  end for
20:  for  $\sigma \in \Sigma$  do
21:    for  $q \in Q$  do
22:       $mask \leftarrow AND(mask_\Sigma[\sigma], mask_Q[q])$ 
23:       $ns \leftarrow XOR(ns, AND(mask, M(q, \sigma)))$ 
24:    end for
25:  end for
26:   $s \leftarrow ns$ 
27: end for
28:  $DFAResult \leftarrow 0$ 
29: for  $ms \in F$  do // ms = match state
30:    $DFAResult \leftarrow XOR(DFAResult, Compare(s, ms))$ 
31: end for
32: return  $DFAResult$ 
```

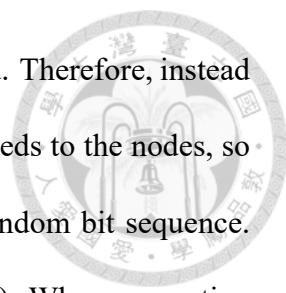


5.3 Optimization 2: Perform Independent-ANDs in One Request

As mentioned in Chapter 4, the bottleneck of the algorithm will be the number of AND operations because all the nodes need synchronization. Therefore, the new strategy of this optimization is to combine multiple independent AND operations and calculate them in one synchronization. One example is that when generating masks, we do *Compare()* on all indexes, and since all the indexes are independent, we can combine them. Consider the scheme of Optimization 1, we will need to perform AND $\log(Q) \times (\Sigma + Q)$ times to generate all masks, and we can combine them so we only need $\log(Q)$ times, where $\log(Q)$ is used to merge the bits and the parameters are correlated so we cannot combine them anymore. Another example is when applying masks to the values, we can also combine them so we only need 1 AND operation to get all results. The improvement and discussion will be in Chapter 6.

5.4 Optimization 3: Use PRNG to Generate Split Beaver Triples

The user has to generate the Beaver triple itself due to privacy concern. In the basic implementation, suppose the required Beaver triple length is l , then the user has to transfer $3 \cdot l \cdot N$ bits to all the nodes in total, where N is the number of nodes, because there are three variables a, b, c in the Beaver triple. If we take a closer look into the *Split()* function, we will find out that $(n - 1)$ of the Beaver triple shares are just randomly generated bits. Further more, $[a]_N$ and $[b]_N$ are also randomly generated bits, and only $[c]_N$ needs to be



calculated once $[a]_1 \sim a_N, b_1 \sim b_N$, and $c_1 \sim c_{(N-1)}$ are determined. Therefore, instead of transmitting all the random bits, we turn to transmit the random seeds to the nodes, so that both the user and the respective node can generate the same random bit sequence. We use AES OFB as the pseudo random number generator (PRNG). When generating random bits, they both setup the same key and IV value, then encrypt null bytes to get the same random bit sequence. With this optimization, the network traffic is then reduced to $l + N \cdot 16 \cdot 2$, where 16 is the length of key and IV value and is actually neglectable because it is much smaller than l . The improvement and discussion will be in Chapter 6.

5.5 Optimization 4: Reduce Input Character Set

The input character set contains 256 characters from `\x00` to `\xff`. If we take a closer look on the DFA matrix, we will find out that the characters can be separated into groups that given a current state, all the characters in the same group lead to the same next state. Therefore, we can choose the biggest group, keep 1 character and remove the rest to reduce the input character set. We can utilize other characters' comparison results to determine whether an input character is in that group or not. The input character is in the group only if all the other comparison results are 0, so we can calculate it by XOR all the other comparison results, and then perform NOT. Since there will be at most one result that is 1 in all the other comparison results, XOR and NOT is actually equivalent to NOR, which produces correct answer yet much faster because XOR and NOT doesn't require synchronization. The improvement and discussion will be in Chapter 6.



Chapter 6

Evaluation

In this chapter, we will analyze our proposed system theoretically and practically. We will evaluate how much does the optimization techniques improve the efficiency, and test the well-optimized system with URLs and patterns of average case to see how the system performs.

6.1 Dataset

It's hard to obtain a malicious URL dataset, due to the Intellectual Property problem, so we use an open-source [ad domain filter list](#) instead [8]. Ad domains share similar characteristics with malicious URLs: they are not welcomed to many people that those people apply filters to avoid access to them, and they tend to leverage some evading techniques to bypass the filters. Those filters are written with specific rules, capable of filtering and removing advertising elements from the HTML of webpages or directly blocking traffics from ad domains. We follow the rules introduced in [7] to translate the filters into regex patterns for further analysis.



6.2 Complexity Analysis

6.2.1 Complexity of DFA Size

First, we focus on analyzing the relationship between the length of regex pattern and the corresponding DFA size (the number of states). This is important because for a regex pattern of length L , the corresponding DFA size may be up to 2^L , which will then be infeasible in the real world [15]. The worst case is, however, considered rare in the real world, especially in the scheme of URL patterns.

To verify the above argument, we analyze whether regex length and DFA size are linearly correlated in most of the URL patterns.

Figure 6.1 shows the relationship between regex length and DFA size. Notice that there are some extreme values, so we first go through all the patterns that produce those extreme values, and the results are as follows:

- (https?://)104\.154\..\{100,\}
- (https?://)104\.197\..\{100,\}
- (https?://)104\.198\..\{100,\}
- (https?://)130\.211\..\{100,\}
- (https?://)142\.91\.159\..\{100,\}
- (https?://)213\.32\.115\..\{100,\}
- (https?://)216\.21\..\{100,\}

- (https://)217\182\11\.{100,}
- (https://)51\195\31\.{100,}



The patterns seem to be incomplete IP filters, while the last part, $\{100,\}$, means to match any characters 100 times or above, which is unreasonable and will never match any IP addresses. A reasonable explanation will be that either there are typos in the patterns, or they are correct filters but only apply in extreme cases. Therefore, we decide to remove them from the dataset. With the rest of the data, we found that the majority of the data matches three linear regression lines, as shown in Figure 6.2, while line 1 seems to be very steep, so we focus on data on line 1 and take a closer look. We filter all the data that $\frac{\Delta y}{\Delta x} > 2$, as shown in Figure 6.3. To confirm their approximate linear relationship, we conducted a cubic regression analysis, and the result shows that the coefficients for higher-order terms are significantly smaller than the coefficient for the first-order term. In most cases, the pattern length is not excessively long (< 200), so the quadratic term is at most of the same magnitude as the linear term. Therefore, it does not affect the linear result.

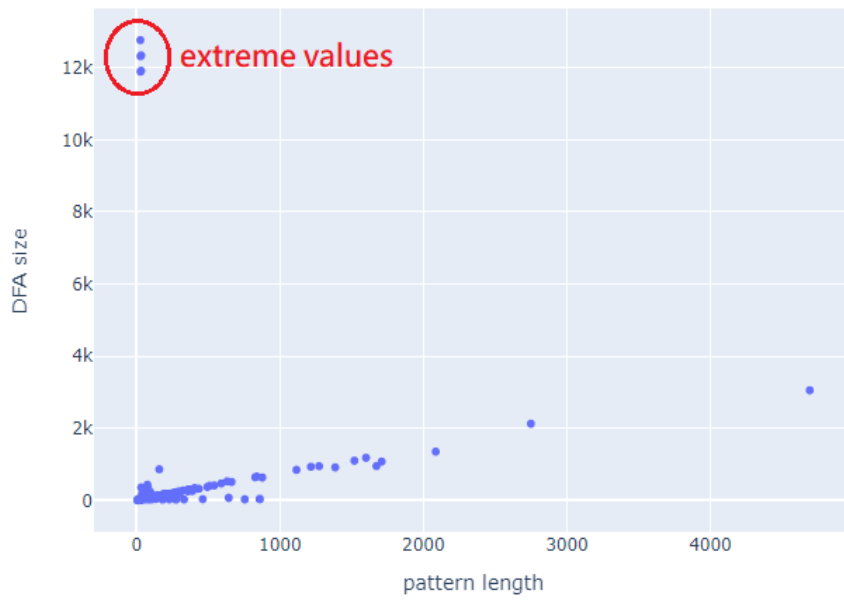


Figure 6.1: DFA sizes of easylist filters

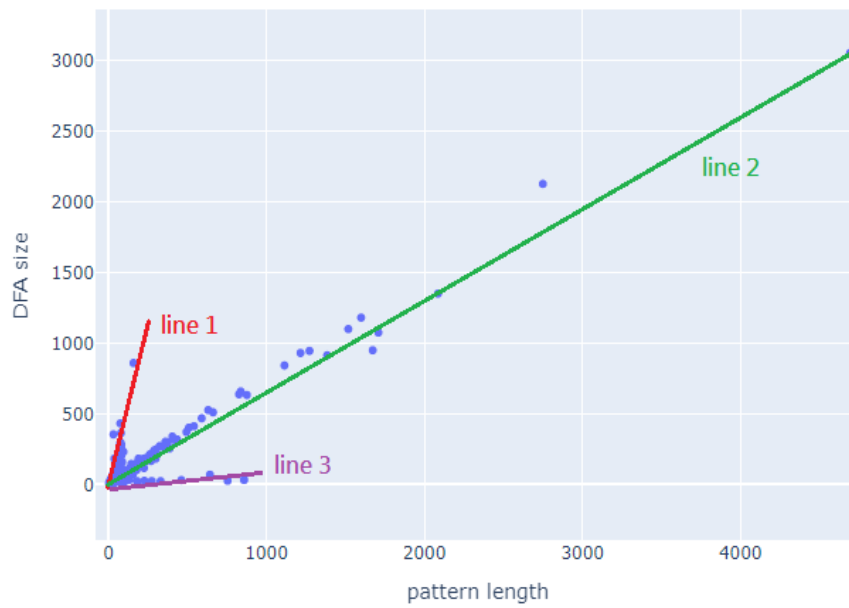


Figure 6.2: DFA sizes of easylist filters excluding extreme values

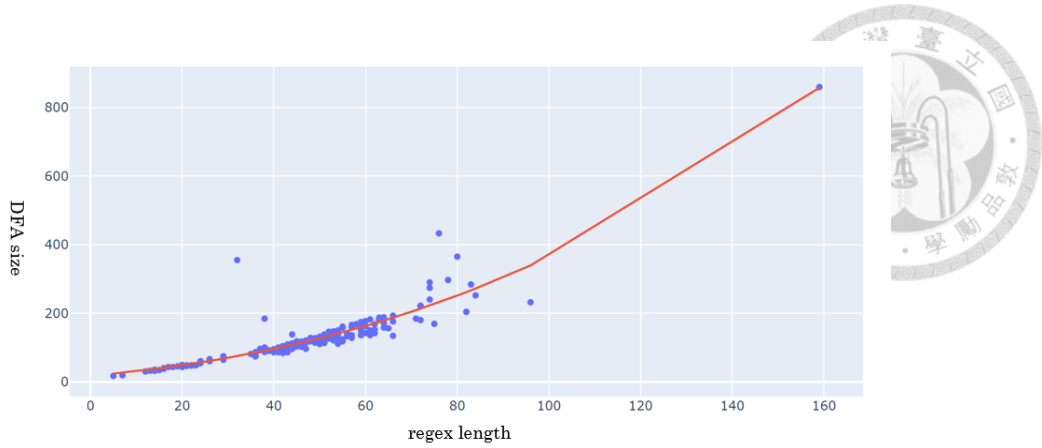


Figure 6.3: DFA sizes of easylist filters excluding extreme values and $\frac{\Delta y}{\Delta x} > 2$. The orange line is the cubic regression result: $y = 1.14x + 1.84 \times 10^{-2}x^2 + 4.83 \times 10^{-5}x^3$

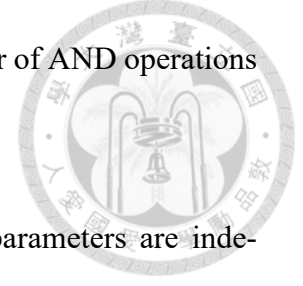
6.2.2 the Number of AND Operations

We analyze the the number of AND operations used in the system because it requires synchronization, which makes it the slowest operation and the bottleneck of the algorithm. The AND operations are used in *Compare()* function and when applying masks to the values.

Algorithm 1 shows the algorithm of our basic implementation. Suppose that the length of input u is L and there are $|Q|$ states. The index of DFA is the combination of input characters and states, so the number of index will be $|Q| \cdot |\Sigma|$, which will require $\log(|Q| \cdot |\Sigma|) = \log|Q| + \log|\Sigma|$ bits to represent an index. For *Compare()*, it will then require $\log(\log|Q| + \log|\Sigma|)$ ANDs to NOR all the bits. For applying mask, we need $|Q| \cdot |\Sigma|$ ANDs to go through all the values. Hence, the number of AND operations required for basic implementation is $L \cdot |Q| \cdot |\Sigma| \cdot (\log(\log|Q| + \log|\Sigma|) + 1)$. Note that we assume that the accept state set F is very small (usually < 10), so the comparison with accept states is not considered here.

Algorithm 2 shows the algorithm applying Optimization 1. In Optimization 1, we separate

the indexes to reduce the number of *Compare()* needed. The number of AND operations thus become $L \cdot [(|Q| \cdot \log(\log|Q|) + |\Sigma| \cdot \log(\log|\Sigma|)) + |Q| \cdot |\Sigma|]$.



In Optimization 2, we concatenate and merge the ANDs whose parameters are independent. Since all the indexes are independent, we are able to compare the indexes, generate masks, and apply masks to values simultaneously, which eliminates the factor of $|Q| \cdot |\Sigma|$ and $|Q| + |\Sigma|$. The number of AND operations required thus becomes $L \cdot (\log(\log|Q|) + \log(\log|\Sigma|) + 1)$.

Optimization 3 and 4 are irrelevant to the number of AND operations, so it remains unchanged.

6.2.3 Beaver Triple Length

We analyze the the Beaver triple length required in the system because the beaver triples are the majority of the network traffic sent by the user. Each bit of AND consumes 1 bit of Beaver triple, which generates 3 bit of traffic because there are three variables in the Beaver triple.

Algorithm 1 shows the algorithm of basic implementation. For *Compare()*, it will require $\log(|Q| + |\Sigma|)$ Beaver triples in total to calculate $\log(|Q|)$ -input NOR. For applying mask, $|Q| \cdot |\Sigma|$ ANDs are needed, and the lenth of mask and values are $\log|Q|$. Hence, the number of Beaver triples required for basic implementation is $L \cdot |Q| \cdot |\Sigma| \cdot (\log(|Q| + |\Sigma|) + \log|Q|)$.The comparison with accept states is, again, not considered here.

Algorithm 2 shows the algorithm applying Optimization 1. In Optimization 1, we separate the indexes to reduce the number of *Compare()* needed, so the required Beaver triples are also reduced. The number of Beaver triples required thus becomes $L \cdot (|Q| \log|Q| +$

$$|\Sigma|\log|\Sigma| + |Q| \cdot |\Sigma| \cdot \log|Q|).$$

In Optimization 2, we only concatenate and merge the independent ANDs, which do not affect the length, so the number of Beaver triples required remains unchanged.

In Optimization 3, we improve the distributing process of Beaver triples, but the number of Beaver triples required remains unchanged.

In Optimization 4, we shrink the input character set Σ to Σ' . Thus, the number of Beaver triples required becomes $L \cdot [(|Q|\log|Q| + |\Sigma'|\log|\Sigma'|) + |Q| \cdot |\Sigma'| \cdot \log|Q|]$.

6.3 Security Analysis

In this section, we will explain why our system preserves both server's and user's privacy. Because all the data is sent to the nodes for the matching process, we focus on how our system preserves privacy against an adversary node. An adversary node may be curious about revealing the server's and the user's secrets, including the DFA matrix, URL and the matching result.

A naive way for the adversary node may be to reconstruct the secrets from the secret shares. To do so, he will need to obtain all the secret shares for reconstruction. However, because at least one node is considered honest and will not collude, the adversary node can obtain at most $(N - 1)$ shares; and since XOR has perfect secrecy if the shares are generated with enough randomness, so missing any piece will result in failure in reconstruction. Therefore, it is impossible for the adversary node to directly reveal the secret.

The adversary node may then try to gather additional information from the matching process and try to reveal the secrets with the additional information. This includes two at-

tempts: revealing the parameters of the algorithm (DFA matrix, URL) by reverse-engineering, and reveal or predict the matching result from the intermediate values.

To reveal the DFA matrix and the URL, the adversary node may utilize the intermediate values of AND operation $r = AND(x, y)$, where the nodes exchange shares and reconstruct x' and y' , which are mentioned in Chapter 2. XOR and NOT, in contrast, do not need synchronization and thus they don't bring any extra information to the adversary node. In our algorithm, a portion of the URL or the DFA may be represented as the x or y in the AND operation, so if the adversary could obtain x from x' (or y from y'), he could reveal the DFA matrix and the URL. However, it is impossible for the adversary node to do so because $x' = x \oplus a$, and the adversary node doesn't have enough shares to reconstruct a . The adversary node may also reverse lookup the next state in the DFA matrix to obtain the list of possible indexes, which contains information of the URL; however, it is also impossible because all the next state are calculated from the algorithm, so the secret share of next state calculated would likely be different from the one in the DFA. Therefore, it is also impossible for the adversary node to reveal the DFA and the URL during the matching process.

The last attempt of the adversary node would be to reveal or predict the matching result. However, this is also impossible because all the intermediate values and the result are on secret sharing domain and, again, the adversary node could not obtain enough shares to reveal them. Besides, all the shares are only random bytes from the node's perspective, so all the intermediate values and the result still has randomness, which makes it hard to predict the result of the matching process. We thus conclude that our system can preserve both server's and user's privacy.



6.4 Computation Time and Network Traffic Size

In this part, we will analyze how the optimizations improve the system and the overall performance of the well-optimized system. All the experiments are done on Ubuntu 20.04 WSL system with AMD Ryzen 4750G CPU and 32G memory, and 2 nodes are used for computing the result. For simplicity, we run all the components on the same computer and use only loopback interface to transmit data between components. We will instead use the traffic size to evaluate the network performance.

Table 6.1: Results of evaluating input = "agg" and regex = ".*g"

Optimization	network(s)	total(s)	network / total	AND op.(#)	traffic(KB)
base	110.06	113.10	97.31%	65298	229.65
O1	22.94	23.61	97.16%	15534	91.02
O2	0.01	0.05	20%	33	91.02
O3	0.01	0.06	16.67%	33	15.26
O4	0.01	0.04	25%	33	7.89

Table 6.1 shows the results of evaluating a simple case, where the regex means to match any string that ends with "g". Network time is the time consumed by python socket.recv() function, which includes data transmission time and synchronization time. Total time is the time consumed by the whole evaluation process. AND operation means the total number of AND operations used by the evaluation process. Traffic means the size of data transmitted by the user to all nodes, which is mainly composed of beaver triples.

We can see that in basic implementation, the network time constitutes the vast majority of the total time. Since it's impossible to take 110 seconds to just transmit less than 1MB of data via localhost network, it implies that synchronization in AND operation is indeed the

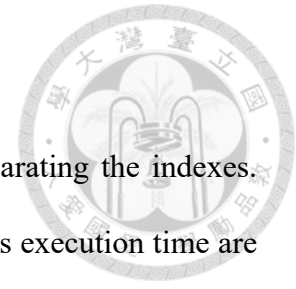
bottleneck.

In Optimization 1, we optimize the way to generate masks by separating the indexes. The result is that less AND operations are used, so less beaver triples execution time are needed. We can see that the number of ANDs are reduced to one-fourth of the original, so the traffic is decreased by about 2.5 times, and the execution time is also reduced by about 5 times. Note that the byte length of each AND is not the same, so the number of AND is not proportional to the traffic.

In Optimization 2, the AND operations are computed in parallel. This optimization only reduce the number of synchronization needed, not the byte length of ANDs, so the traffic, which is the beaver triples required, remain the same. However, we still observe significant reduction in execution time, as we successfully reduce the number of ANDs (number of synchronization) by about four to five hundred times.

In Optimization 3, we optimized the way to transmit beaver triples by utilizing PRNG. The traffic is further reduced by about 6 times, while the execution time remain the same because the algorithm is not changed.

In Optimization 4, we reduce the input character set Σ by eliminating characters that have same next states. Because the execution time becomes very short, the effect of the error also becomes more significant. The extent of performance improvement depends on the reduction in the number of characters, which varies with each DFA. In this example, the execution time is further reduced and the traffic is also reduced by about 2 times. The average number of reduced chars will be evaluated in next experiment.



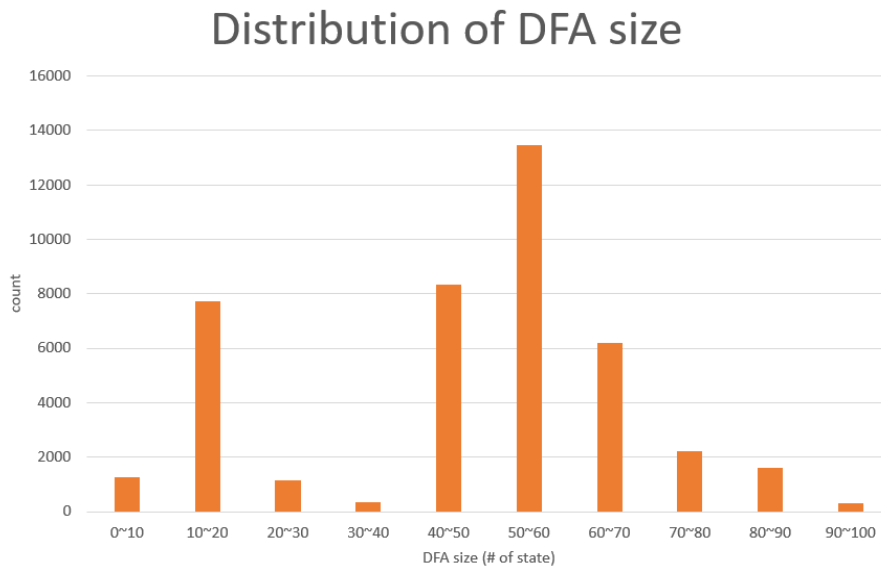


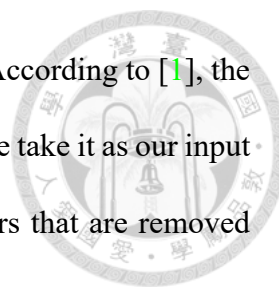
Figure 6.4: Distribution of DFA size: 98.98% of URL patterns (43021) with less than 100 states

Table 6.2: Average result of evaluating input length = 66

$ Q $	reduced char(#)	node computing time(s)	user computing time(s)	traffic(KB)
0~10	248.98	0.52	0.005	7.71
11~20	236.44	0.63	0.01	26.88
21~30	214.52	0.83	0.03	80.70
31~40	188.24	1.38	0.07	202.23
41~50	117.66	2.74	0.18	504.44
51~60	113.14	3.70	0.21	611.11
61~70	113.88	3.90	0.25	701.70
71~80	110.76	4.62	0.28	817.39
81~90	107.14	5.75	0.32	940.09
91~100	108.26	6.50	0.36	1053.39

Table 6.2 shows how the computing time and traffic vary as the number of states increases.

Here we ignore the patterns whose corresponding DFA size > 100 , because the majority of the patterns(98.56%) has its DFA size less than or equal to 100. We categorize the rest of patterns into 10 groups, randomly sample 50 patterns from each group, and record



the average result. Figure 6.4 illustrates the distribution of the data. According to [1], the average URL length for a top 10 result in Google is 66 character, so we take it as our input length. The number of reduced char means the number of characters that are removed from Σ (originally 256 characters) by Optimization 4.

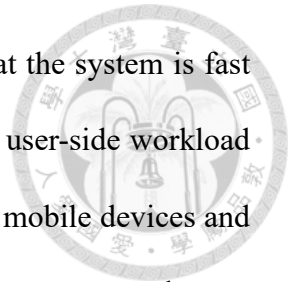
We can see that as the DFA size becomes bigger, the pattern tends to be more complicated, so the characters that can be removed from Σ becomes less, but we can still remove about 110 characters in average for large DFAs, which brings us about 1.75X performance improvement. We also found that in average case of about 50~60 states, the matching process takes only about 3.7s, which is very fast; even for the biggest DFA tested (90~100 states) - which is very big and rare, considering that it is transformed from only one URL pattern - the matching process takes only about 6.5s to evaluate the result, which is good enough for extreme cases.

Table 6.3: Result of evaluating DFA size = 54

URL length	node computing time(s)	user computing time(s)	traffic(KB)
30	1.74	0.09	267.20
40	2.39	0.13	356.23
50	3.00	0.16	445.26
60	3.56	0.19	534.29
70	4.13	0.22	623.33
80	4.75	0.25	712.36
90	5.54	0.29	801.39

Table 6.3 shows how the computing time and traffic vary as the length of input URL increases. Because there are extreme values in the dataset, we choose the median of the DFA size for the test. We found that in average case which the length of URL is about 60~70, the matching process takes about 3.85s, and for long URLs (90 chars), the match-

ing process takes about 5.5s to evaluate the result, which implies that the system is fast enough and acceptable. In addition, one may argue that whether the user-side workload is small enough for devices with low computation resources, such as mobile devices and IoT devices. Therefore, we also test how much time it cost for the user to generate beaver triples and split data, which are the main workloads for the user. From Table 6.2 and 6.3, we can see that the user's computing time are all much smaller than the nodes' computing time, which implies that the computing nodes indeed cover most of the workloads and the user's workload is very small compared to the nodes' workload.





Chapter 7

Related Work

Google Safe Browsing [11] is the biggest URL checking service in the world, and its privacy-preserving solution is its Update API [12]. Update API use hash prefix to avoid tampering user's privacy: the server keeps a database of malicious URL hashes, and the user query for a hash prefix and compare it locally with its URL. However, the solution has its limitations. First, the hash prefix can only be used on exact matching, so even with URL canonicalization, it still has limited applicability. Second, It is pointed out that even Update API has its privacy issue [9]. Because of canonicalization, sometimes the hash prefix may leak information of the URL. What makes it worse, a malicious service provider may even implant hash prefixes of specific websites into the user's local database to track the user's browsing histories of those websites.

Some works aim to solve the privacy issue of Google Update API. Cui et al. proposed PPSB [4] which downloads encrypted URL hash list to the user's device instead and utilize oblivious pseudorandom function (OPRF) to search and decrypt the result for the user. Du et al. proposed PEBA [6] that utilize a proxy server and trusted hardware to avoid downloading encrypted URL hash list to the user's device, which consumes the user's

storage. Both works are based on Update API, and therefore they both don't support regex matching.



In the domain of privacy-preserving regex matching, another application is Privacy-Preserving Intrusion Detection System (PPIDS). Niksefat et al. proposed ZIDS [21] which leverage ODFA protocol to match snort rules on traffic data, while we focus on outsourcing the computation to the server-side so our user has less workloads and smaller data transmitted. Sgaglione et al. [23] proposed another approach with Fully Homomorphic Encryption (FHE), but the efficiency is very low and the functions are limited.

DNA and chemical compound matching also have privacy-preserving version, though their data may be in special form instead of an ASCII string. Nakagawa et al. [20] proposed a privacy-preserving substring matching system for DFA sequence, which leverage computing nodes to help evaluating comparison results on share secrets, and is the inspiration of our proposed system. Kim et al. [17] proposed a FHE search on SQL database which supports single wildcard character (*) and NOT (!). Shimizu et al. [24] proposed a privacy preserving search for chemical compound databases with fuzzy matching based on Tversky index. All the above solutions are special-case solutions and are not suitable in URL checking scheme.



Chapter 8

Discussion and Future Work

In this chapter, we discuss some issues of the system that are not out of scope in this paper, some of which may be studied in the future work.

8.1 Verifiability

In this paper, we assume that all the nodes will follow our algorithm and return the correct result shares. However, it's good to have some mechanisms for the user to verify that whether the result received from the nodes is correct or not. Some naive solutions may be to send the request to two sets of nodes, or split the URL in a different way and send the request again, then see if the results are the same or not. However, either relying on more nodes or doubling the evaluation time is not a good solution to this problem. An alternative way is that the nodes somehow prove themselves that the computations are indeed done on the shared secrets. There are several works about how the server can verify the input, but currently, to the best of our knowledge, no work that is able to verify the computation on share secrets have been done. How to convince the user the result is generated from

correct calculation is left for future work.



8.2 Availability

The original Shamir's secret sharing allows people to reconstruct the secret data with only t pieces of share secrets, where $t \leq N$ and N is the total amount of share secrets, which leaves a room for fault tolerance. However, to simplify the Computation process, we let $t = N$ in our system, which result in zero fault tolerance. Besides, the nodes need to be synchronized during computation to evaluate the result, which means that once a node is down, the evaluation process is forced to termination. In such situation, the user can assign a new node to proceed the process by sending the URL secret share as same as the broken node possess. the node then asks for all the intermediate values from all the other nodes that have already been calculated and broadcasted to quickly catch up and continue the evaluation process.

8.3 Evaluation on Multiple DFAs

In previous chapters, we only match one pattern with the URL. In practice, the server will have a set of patterns to be matched with the user's URL. How to merge DFAs in an efficient way is an important issue. Liu et al.[18] proposed **RegexGrouper** in 2014, which helps grouping DFAs and merging them group by group with less states. On the other hand, the user may also assign more nodes to parallel-computing the DFAs to speedup the evaluation process.

8.4 Limitation



In our proposed system, we utilize google RE2[14] engine to transform regex patterns into DFA matrix. RE2 supports all basic regex functions, but not all extended regex functions. For example, in Perl Compatible Regular Expressions(PCRE), there are back-reference and look-around assertion which are not supported by RE2 due to the difference of implementation - PCRE engines mostly use backtracking technique while RE2 use Finite Automata to implement. On the other hands, our system only support signature-based URL checking, while there are other technologies to achieve URL checking, such as ML-based detection. There may be other techniques to achieve privacy-preserving machine learning, such as Homomorphic Encryption, but they are out of scope and thus not considered here.



Chapter 9

Conclusion


In this paper, we build a privacy-preserving URL checking system by introducing computing nodes to evaluate results on secret shares. As long as one of the computing nodes is considered honest, the secrets cannot be reconstructed and thus the privacy of the user and the server are protected. We propose an algorithm to perform regex matching on secret shares, and optimizations that can improve the overall efficiency so that the system can finish the matching process in short period of time. Besides, due to the support for regex matching, our system also has broader privacy-preserving cloud applications, such as pattern-based cloud IDS, Firewalls, and other services.


Our system is developed in Python, which is just a proof of concept; it can be even more efficient if we develop it in C++ with better design and memory management. Compared to prior work (ZIDS), we reduce the computation loading of the user, which is important because the user tend to have less computing power than server or computing nodes; we also reduce the network traffic to further reduce the loading of the user. Our system takes about 3 ~4 seconds for the matching process and about 600KB of network traffic in average case, which is quite efficient.



Bibliography

- [1] Backlinko. Backlinko: google search results analysis. <https://backlinko.com/search-engine-ranking>, May 2023.
- [2] D. Beaver. Efficient multiparty protocols using circuit randomization. *CRYPTO (1991)*, pages 420–432, 1991.
- [3] Ross Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). <https://swtch.com/~rsc/regexp/regexp1.html>, May 2023.
- [4] Helei Cui, Yajin Zhou, Cong Wang, Xinyu Wang, Yuefeng Du, and Qian Wang. Ppsb: An open and flexible platform for privacy-preserving safe browsing. *IEEE Transactions on Dependable and Secure Computing*, 18(4):1762–1778, 2021.
- [5] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In Matt Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX, 2004.
- [6] Yuefeng Du, Huayi Duan, Lei Xu, Helei Cui, Cong Wang, and Qian Wang. Peba: Enhancing user privacy and coverage of safe browsing services. *IEEE Transactions on Dependable and Secure Computing*, pages 1–15, 2022.

- 
- [7] EasyList. Easylist filter blocking rules. <https://adblockplus.org/filter-cheatsheet>, May 2023.
- [8] EasyList. Easylist filter list, version: 202305230654. <https://easylist.to/easylist/easylist.txt>, May 2023.
- [9] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. A privacy analysis of google and yandex safe browsing. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 347–358, 2016.
- [10] V M Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1, oct 1961.
- [11] Google. Google safe browsing. <https://safebrowsing.google.com/>, May 2023.
- [12] Google. Google safe browsing api. <https://developers.google.com/safe-browsing/v4>, May 2023.
- [13] Google. Google safe browsing statistics. <https://transparencyreport.google.com/safe-browsing/overview>, May 2023.
- [14] Google. Google/re2: Re2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in pcre, perl, and python. it is a c++ library. <https://github.com/google/re2>, May 2023.
- [15] Hermann Gruber and Markus Holzer. From finite automata to regular expressions and back — a summary on descriptonal complexity. *Electronic Proceedings in Theoretical Computer Science*, 151:25–48, may 2014.

- 
- [16] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
- [17] Myungsun Kim, Hyung Tae Lee, San Ling, Benjamin Hong Meng Tan, and Huaxiong Wang. Private compound wildcard queries using fully homomorphic encryption. *IEEE Transactions on Dependable and Secure Computing*, 16(5):743–756, 2019.
- [18] Tingwen Liu, Alex X. Liu, Jinqiao Shi, Yong Sun, and Li Guo. Towards fast and optimal grouping of regular expressions via dfa size estimation. *IEEE Journal on Selected Areas in Communications*, 32(10):1797–1809, 2014.
- [19] Edward F Moore et al. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.
- [20] Yoshiki Nakagawa, Satsuya Ohata, and Kana Shimizu. Efficient privacy-preserving variable-length substring match for genome sequence. *Algorithms for Molecular Biology*, 17(1):9, Apr 2022.
- [21] Salman Niksefat, Babak Sadeghiyan, Payman Mohassel, and Saeed Sadeghian. Zids: A privacy-preserving intrusion detection system using secure two-party computation protocols. *The Computer Journal*, 57(4):494–509, 2014.
- [22] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.
- [23] Luigi Sgaglione, Luigi Coppolino, Salvatore D’Antonio, Giovanni Mazzeo, Luigi Romano, Domenico Cotroneo, and Andrea Scognamiglio. Privacy preserving intrusion detection via homomorphic encryption. In *2019 IEEE 28th International*

Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pages 321–326, 2019.



- [24] Kana Shimizu, Koji Nuida, Hiromi Arai, Shigeo Mitsunari, Nuttapong Attrapadung, Michiaki Hamada, Koji Tsuda, Takatsugu Hirokawa, Jun Sakuma, Goichiro Hanaoka, and Kiyoshi Asai. Privacy-preserving search for chemical compound databases. *BMC Bioinformatics*, 16(18):S6, Dec 2015.
- [25] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, jun 1968.