Privacy-Preserving Regular Expression Matching using Nondeterministic Finite Automata

Ning Luo
Northwestern University
ning.luo@northwestern.edu

Chenkai Weng
Northwestern University
ckweng@u.northwestern.edu

Jaspal Singh
Oregon State University
singjasp@oregonstate.edu

Gefei Tan
Northwestern University
gefeitan@u.northwestern.edu

Ruzica Piskac
Yale University
ruzica.piskac@yale.edu

Mariana Raykova[*]
Google Inc.
marianar@google.com

May 5, 2023

Abstract

Motivated by the privacy requirements in network intrusion detection and DNS policy checking, we have developed a suite of protocols and algorithms for regular expression matching with enhanced privacy:

- A new regular expression matching algorithm that is oblivious to the input strings, of which the complexity is only $O(mn)$ where m and n are the length of strings and the regular expression respectively. It is achieved by exploiting the structure of the Thompson nondeterministic automata.
- A zero-knowledge proof of regular expression pattern matching in which a prover generates a proof to demonstrate that a public regular expression matches her input string without revealing the string itself.
- Two secure-regex protocols that ensure the privacy of both the string and regular expression. The first protocol is based on the oblivious stack and reduces the complexity of the state-of-the-art from $O(mn^2)$ to $O(mn \log n)$. The second protocol relies on the oblivious transfer and performs better empirically when the size of regular expressions is smaller than 2^{12}.

We also evaluated our protocols in the context of encrypted DNS policy checking and intrusion detection and achieved 4.5X improvements over the state-of-the-art. These results also indicate the practicality of our approach in real-world applications.

1 Introduction

A pattern matching algorithm takes as input a string and a pattern and checks whether the given pattern appears in the string. It has been widely used in many areas, including bioinformatics [Ben99], database search engines [BKS02], intrusion detection systems [VL06], text processing [Kuk92], and digital forensics [Bee09]. Matching patterns are commonly represented as regular expressions because they are able to match text using a structured pattern and are commonly used in search engines, word processors, and programming languages. In this paper we focus on two different aspects of the regular expression matching

[*]The author took part in and made contributions to this work during her time at Yale.

problem. Our research is motivated by two applications that demand distinct solving techniques: monitoring and enforcing DNS query policies, and detecting network intrusions. DNS queries can be monitored for banned URLs and rejected by network administrators, leveraging the expressiveness of regular expressions with low false positive rates and high expressiveness [Moc87; APD+10; XYA+08]. Intrusion detection systems (IDS) can also use regular expressions to identify potential malicious network packets, reducing the task of detecting network attacks to a pattern-matching problem for incoming packets [SP03].

Many of these applications have strict privacy requirements. In the DNS setting, exposing the DNS queries to the network administrator would violate the privacy of users. Enforcing the policy check by examining all outbound packets also prevents the deployment of recent privacy-preserving DNS techniques such as DNS-over-HTTPS [HM18]. However, in this case, the policy (e.g. blocklist) may not need to be kept secret. For example, under the Children's Internet Protection Act, schools are suggested to censor Internet browsing from the school network but should make public the criteria used to block websites [McC04; GAZ+21]. On the contrary, the network intrusion detection example would prefer keeping the secrecy for both the input string and pattern. The users tend to keep packets secret from IDS. Meanwhile, the IDS may view its regular expression database as its intellectual property, so there is a strong motivation for them to limit the database's access [SLPR15]. In summary, solving a pattern-matching problem in privacy-preserving settings is becoming an increasingly important problem, while the privacy requirements vary across different use cases.

The goal of our research is privacy-preserving regular expression matching. We first consider the settings where the pattern is publicly known. In these settings, a prover holds a private string and generates a proof, which is used to validate if that string is accepted or rejected by the given public pattern. It is required that the proof reveals no information about the input string. We refer to this as *ZK-regex* and realize it by using zero-knowledge proofs (ZKP) [GM84]. A typical application is the blocklisting of DNS queries demonstrated by zero-knowledge middlebox (ZKMB) [GAZ+21]. We next consider the settings where a pattern holder and a string holder want to verify whether the pattern matches the string. Both parties should not learn anything beyond the result (true or false) and the length of inputs. We refer to this as *secure-regex* and realize it by secure two-party computation (2PC) protocols [Yao82]. A typical application is the privacy-preserving network intrusion detection described before.

Challenges. Both ZKP and 2PC securely process an algorithm in the circuit model by first translating any algorithm into an oblivious version. This means the access pattern of the underlying algorithm should be independent of inputs except for their length. However, regular expression pattern matching in plaintext relies on finite automata simulation, and the known algorithms and data structures are highly dependent on the input of regular expressions and strings. If the access pattern of evaluation in the plaintext leaks inputs, as demonstrated in the following examples, additional oblivious data structures or algorithms can be used to enhance privacy. Existing works [SHd+14; LW13] utilize matrices to represent finite automata and performs linear scans over the matrices for simulation to achieve this. However, this approach introduces overhead that is quadratic to the size of the finite automata and regular expressions, which can make the protocol relatively impractical.

1.1 Our Main Observation

We have developed a novel algorithm for pattern matching that employs linear scans over finite automata for simulation. Specifically, our approach utilizes Thompson nondeterministic finite automata (TNFA), a type of finite automata commonly employed in regular expression pattern matching. Our key insight is that TNFA can be regarded as a special sparse graph wherein all accessible nodes can be reached via a path with at most one backward edge. This observation enables our algorithm to simulate TNFA while maintaining an access pattern independent of the input string. As a result, our approach is particularly suitable for ZK-regex scenarios. Nonetheless, the access pattern still relies on the TNFA graph, which reveals information about the regular expression from the pattern holder. To address this issue, we later propose protocols for achieving secure-regex scenarios in which the TNFA graph's information is protected using oblivious transfer or oblivious stack, which leads to the most practical performance yet known.

An illustrative example. Before introducing our protocols, we first describe through an example how to do efficiently regular expression matching in plaintext. Thompson proposed a method [Tho68], in which the regular expression is converted into a TNFA with a size linear in the size of the expression (cf. Section 2.1). TNFAs are directed graphs with states as nodes and transitions as labeled edges. This representation reduces pattern matching to finding a valid path from the initial state to an accepting state in the graph for the input string. Thompson's algorithm constructs a set $\mathcal{Q}_i$ of all reachable nodes through valid paths for the first i characters of the input string. If the pattern in the regular expression matches the input string, an accepting state should be in $\mathcal{Q}_m$ where m is the length of the input string. We use m and n to denote the string and pattern length throughout this paper. We demonstrate the TNFA simulation for pattern matching in regular

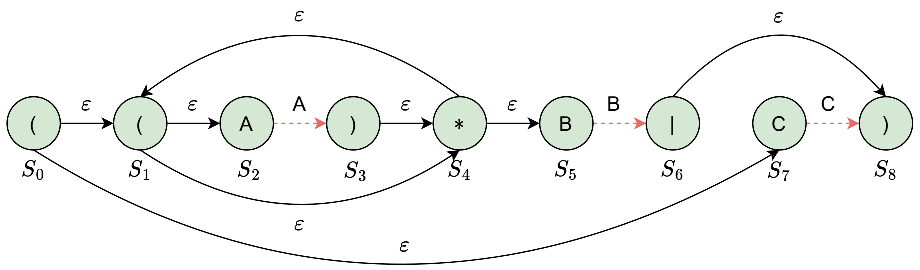

Figure 1: Example TNFA for regular expression re $= ((A)^*B|C)$. The solid arrows are directed edges in G and define epsilon transitions of the TNFA. The dashed arrows are the character transition.

expressions using the example in Figure 1. The figure shows a TNFA for the regular expression $((A)^*B|C)$ where S_0 is the initial state and S_8 is the accepting state. For the input string AB, which matches $((A)^*B|C)$, there is a valid path $S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{\varepsilon} S_2 \xrightarrow{A} S_3 \xrightarrow{\varepsilon} S_4 \xrightarrow{\varepsilon} S_5 \xrightarrow{B} S_6 \xrightarrow{\varepsilon} S_8$. The path is considered valid as the labels along the path from the sequence $\varepsilon^* A \varepsilon^* B \varepsilon^*$, where the superscript '$*$' indicates any number of repetitions. The following steps explain the process of determining if the given TNFA accepts AB:

1. Set $\mathcal{Q}_0$ is initialized to $\{S_0\}$, and then all states reachable from $\mathcal{Q}_0$ through ε-labeled edges are added. This results in $\mathcal{Q}_0 = \{S_0, S_1, S_2, S_4, S_5, S_7\}$.
2. As $S_2 \in \mathcal{Q}_0$ and there is an edge (S_2, S_3) labeled with A, set $\mathcal{Q}_1$ is initialized to S_3 for the first character A in the string AB. This is then extended to $\mathcal{Q}_1 = \{S_1, S_2, S_3, S_4, S_5\}$ by including states reachable through ε-labeled edges.
3. The same is repeated for the second character B in the string AB, resulting in $\mathcal{Q}_2 = \{S_8\}$.
4. Since the accepting state S_8 is in $\mathcal{Q}_8$, there is a valid path in the TNFA from S_0 to S_8 for AB, meaning $((A)^*B|C)$ matches AB.

The example demonstrates the two-step process of constructing each set $\mathcal{Q}_i$: initializing $\mathcal{Q}_i$ using $\mathcal{Q}_{i-1}$ and character-labeled edges, and then extending $\mathcal{Q}_i$ by adding states reachable through ε-labeled edges. The first step is called the *character transition* and the second step is the *epsilon transition*. Note, however, that from the sequence of sets $\mathcal{Q}_0, \mathcal{Q}_1, \ldots$ one can reconstruct both, the input string and the TNFA. In this paper, we develop efficient protocols for regular expression matching, based on the above TNFA simulation, while maintaining privacy of input.

Protocol	Comm. Complexity	Round Complexity
[Ker06]	$O(m2^n\lvert\Sigma\rvert$	$O(m)$
[Fri09]	$O(m(2^n + \lvert\Sigma\rvert))$	$O(1)$
[MNSS12]	$O(m2^n\kappa)$	$O(1)$
[LW13]	$O(\kappa mn(n\lvert\Sigma\rvert))$	$O(m)$
[SHd$^+$14]	$O(\kappa n^2(m + \lvert\Sigma\rvert))$	$O(mn)$
Ours (from OS)	$O(\kappa mn(\log n + \log\lvert\Sigma\rvert)$	$O(1)$
Ours (from OT)	$O(mn(n + \kappa\log\lvert\Sigma\rvert))$	$O(mn)$

Table 1: Comparisons with existing secure-regex protocols. m, n and Σ are the length of the string, the size of the regular expression, and the alphabet, respectively. Protocols [Ker06; MNSS12; Fri09] exhibit complexity exponential to the size of the regular expressions because they are DFA-based.

1.2 Our Contribution

The pattern matching in plaintext consists of character transition and epsilon transition (details in Section 3). In privacy-preserving settings, the character transition can be executed directly through linear scans and equality checks, which are easily handled by ZKP or 2PC protocols. Thus, the main challenge of this work is a design of a data-oblivious epsilon transition. Epsilon transitions involve searching for all reachable states in the TNFA. In the plaintext execution, this search can be performed using a depth-first search (DFS). However, the best-known oblivious DFS for general graphs requires a circuit size of $O(n^2)$, leading to a total complexity of $O(mn^2)$ to process a length-n regular expression and a length-m string [BSA13]. This complexity is not feasible for practical, real-world applications.

TNFA simulation via two linear scans. Our proposed pattern-matching algorithm uses two linear scans for epsilon transition and takes advantage of the fact that the TNFA can be modeled as a sparse graph. Specifically, we observed that all reachable nodes in the TNFA can be accessed via a path with at most one backward edge in the epsilon transition. This insight enables us to derive the elements of the set $\mathcal{Q}_i$ by performing two passes of linear scans. Importantly, the access pattern of the epsilon transition is independent of the input string.

ZK-regex. We present a ZK-regex protocol that enables a prover to prove whether a public regular expression matches a string without revealing any information about the string. Define $\log|\Sigma|$ to be the bit length of the alphabet. The circuit for our ZK-regex protocol is derived from our two-linear scan algorithm and is of size $O(mn\log|\Sigma|)$. We leverage the fact that our TNFA simulation algorithm is already oblivious to input strings so that no additional data-oblivious algorithm is needed to protect the access patterns. Our ZK-regex circuit fits into any general-purpose ZKP framework. In addition, we apply ZK-regex to examining encrypted Transport Layer Security (TLS) packets. It allows a packet sender to prove whether a specific pattern matches her packet without exposing the packet's content, enabling effective filtering of malicious traffic while preserving user privacy.

Secure-regex. To minimize the need for additional data-oblivious operations to protect the graph structure, we further investigate the properties of TNFAs and prove a bound on the maximum degree of each node. By limiting the number of predecessors and successors of each node, we develop two efficient two-party protocols for secure-regex. Define κ to be the computational security parameter and σ to be the alphabet. The first protocol, based on the oblivious stack (OS), is a constant-round protocol with a communication complexity of $O(\kappa mn(\log n+\log|\Sigma|))$, outperforming the previous best-known result of $O(\kappa n^2(m+|\Sigma|))$ [SHd$^+$14]. The second protocol, based on 1-out-of-n+1 oblivious transfer (OT), achieves a communication overhead of $O(mn(n + \kappa\log|\Sigma|))$ with a linear number of round-trips while incurring a higher asymptotic complexity.

However, it outperforms the first protocol in scenarios with low network latency and short input regular expressions. Comparisons with related work on both asymptotic complexity (Table 1) and empirical performance (Figure 10) demonstrate that our approach achieves both asymptotic and empirical optimality for secure-regex.

Implementation and evaluation. To empirically evaluate our proposed protocols, we implemented ZK-regex over TLS and measured the proof size and prover time. We demonstrate the practicality of our ZK-regex protocol by running our implementation on a set of regular expressions used for DNS filter Pi-hole [pi-22]. For a 128-byte string, a proof for the longest pattern in Pi-hole (97 bytes) can be generated in 0.57s with a size of 397KB. Our secure-regex protocols attain the best-known results for two-party regular expression matching, and their practicality is demonstrated in real-world intrusion detection based on SNORT PCRE [CIS]. The OT-based protocol performs matching on a 512-byte string and a 63-byte pattern in 4s in a LAN with 100Mbps bandwidth. In a WAN with 60ms latency, our OS-based protocol performs matching on a 512-byte string and a 15-byte pattern in 4s. We conducted a comparison of our secure-regex implementations that utilize OS-based and OT-based protocols, testing them under varying network and input conditions. Additionally, we compared our OT-based protocol secure-regex against that of [SHd$^+$14]. Our findings indicate that our OS-based protocol reduces communication costs by 4.5X.

1.3 Related Work

ZK-regex. To the best of our knowledge, the only instantiation of zero-knowledge regular expression pattern match is demonstrated in a work by Franzese et al. [FKL$^+$21]. Their protocol is based on a constant-overhead oblivious array access in the ZK setting. It requires m access to a size-$O(2^n)$ array. We provide more detailed comparison in Section 6.2. For other work, both [SW$^+$21] and [KM14] mention regular expression pattern matching in their ZK algorithms without instantiating this functionality. Toots et al. design a ZK algorithm that proves whether a string is accepted by a nondeterministic finite automata (NFA) [Too]. However, it does not support private epsilon transition thus its application is limited.

In the application level, there exists prior work that also study the ZKP with input the TLS messages. Zero-knowledge middleboxes (ZKMB) [GAZ$^+$21] studies encrypted DNS-over-TLS policy checking via zk-SNARK. It utilizes a non-membership proof to prove that an encrypted URL is not on a blocklist. The work of DECO [ZMM$^+$20] builds a decentralized oracle which allows a user to prove to third parties the provenance of data encrypted in a TLS session. Our work on regex pattern matching enriches the policy check methods in these work. For example, ZKMB uses Merkle tree non-membership proofs to prove that a domain is not in a public blocklist [GAZ$^+$21]. Though the blocklist-based system is straightforward and efficient, studies show that this coarse-grained approach does not offer accurate classification [YMMP16]. It is common for the harmless web applications and malicious ones to have the same domain suffix, thus creating false alarms. To solve this issue, the blocklist based systems are evolving into the control system based on regular expressions which offers finer-grained filtering. This is the main reason that our regular expression based pattern matching algorithms are more suitable for zero-knowledge policy checks, as it provides lower false positive rates and higher expressiveness.

Secure-regex. Secure protocols for general NFA evaluation have been proposed in [LW13; SHd$^+$14]. They either employ computationally-intensive homomorphic encryption or have a communication complexity of $\Omega(mn^2\kappa)$ (where κ is the security parameter), which makes them less practical than our protocols. Although the code and empirical evaluation results of [LW13] are not accessible, we have compared the communication cost of [SHd$^+$14] with our protocols and the results are demonstrated in Figure 1.

Protocols in [Ker06; TPKC07; Fri09; MNSS12; LW13] involve the private evaluation of deterministic finite automata (DFA). However, their complexity is proportional to the size of the DFA, which can be exponential in the size of the regular expression. he size of Nondeterministic Finite Automata (NFA) and

Deterministic. We recommend consulting the empirical evaluation and comparison presented in [SHd+14] on the size of DFA and NFA for the same regular expression.

A comparison of the complexity of regular expression pattern-matching protocols can be found in Table 1. Numerous secure two-party pattern matching protocols have been proposed, as described in [BEDM+13; YSK+13; Ver11; HL08; KRT18; DA18; SBB19]. These protocols facilitate string operations such as exact matching, substring matching, approximate matching, and wildcard matching. In contrast, our methods enable the use of more expressive languages, i.e., regular expressions, to specify patterns. Moreover, the utilization of data-oblivious graph algorithms as presented in [BSA13; WNL+14] can be extended to simulate TNFAs in a secure two-party computation setting, but this approach results in quadratic complexity relative to the length of the regular expressions.

2 Preliminaries

2.1 Regular Expressions and Thompson NFAs

In this section, we introduce formal definitions for regular expressions and for NFAs. We present Thompson's construction [Tho68], which shows how any regular expression can be converted into an equivalent NFA that accepts the same set of strings as the regular expression does.

Regular expression. A regular expression (regex) on the alphabet Σ consists of characters in Σ and meta characters $\{\ |, (,), *\ \}$. Let ε and ϕ denote the empty string and the empty set, respectively. A regular expression re describes a set of strings (or language) $L(\mathsf{re})$ over Σ recursively:

1. basic case: $L(\phi) = \phi$, $L(\varepsilon) = \{\varepsilon\}$ and $L(a) = \{a\}$ for $a \in \Sigma$;
2. concatenation: $L(\mathsf{re}\ \mathsf{re}') = \{xx' | x \in L(\mathsf{re}), x' \in L(\mathsf{re}')\}$;
3. union : $L((\mathsf{re}|\mathsf{re}')) = L(\mathsf{re}) \cup L(\mathsf{re}')$;
4. loop: $L((\mathsf{re})^*) = \bigcup\limits_{k\in\mathbf{N}} \{x_0x_1\cdots x_k | x_i \in L(\mathsf{re})\} \cup \{\varepsilon\}$.

Nondeterministic finite automata. A nondeterministic finite automata (NFA) N is defined as a 5-tuple $(\mathcal{Q}, \Sigma, \delta, S_0, \mathcal{Q}_A)$. $\mathcal{Q}$ is a set of states and Σ is the alphabet set. $\delta : \mathcal{Q} \times (\Sigma \cup \{\varepsilon\}) \to 2^S$ is a transition function that maps a state along with character to a set of states. For states S, S' we write $S \xrightarrow{\varepsilon} S'$ if either $S = S'$ or there exists a sequence of states $S_1, \ldots, S_n$ with $S = S_1$ and $S_n = S'$ such that for all $1 \le i < n$, $S_{i+1} \in \delta(S_i, \varepsilon)$. For $x \in \Sigma$ and states S, S', we write $S \xrightarrow{x} S'$ when there are states S_1 and S_2 such that $S \xrightarrow{\varepsilon} S_1$ and $S_2 \in \delta(S_1, x)$ and $S_2 \xrightarrow{\varepsilon} S'$. An NFA N accepts a string $x_0 \cdots x_{m-1}$ on Σ if there exists a sequence of states $S_0, S_1, S_2, \ldots, S_m$, such that 1) S_0 is the starting state; 2) for $0 \le i \le m-1$, $S_i \xrightarrow{x_i} S_{i+1}$; and 3) $S_n \in \mathcal{Q}_A$ is an accepting state.

Thompson NFA construction. An NFA, N, is equivalent to a regular expression, re, if $L(N) = L(\mathsf{re})$. It is well known that any regular expression can be converted to an equivalent NFA using Thompson's construction, which is called a Thompson NFA (TNFA). We adopt a specific variant [SW11] of the TNFA construction, which we explain in Algorithm 8 in the appendix. A characteristic of a TNFA denoted by such G is that all edges within the alphabet Σ, denoted by E_Σ, are of the form $(j-1, j)$, where j is a specific integer.

Given a regular expression, re, the algorithm processes each character in re in order and outputs a directed graph, $G = (V, E = E_\Sigma \cup E_\varepsilon)$ where $V = \{0, \cdots, n\}$. G can be interpreted as a TNFA as follows:
- $\mathcal{Q} = V$, $S_0 = 0$ and $\mathcal{Q}_A = \{n\}$;
- $j \in \delta(i, \varepsilon)$ if $(i, j) \in E_\varepsilon$;
- $j \in \delta(i, c)$ if $(i, j) \in E_\Sigma$ is labeled by $c \in \Sigma$;
- $\delta(i, c) = \phi$ for the rest of the undefined $(i, c) \in V \times (\Sigma \cup \varepsilon)$.

By this way, the graph $G = (V, E = E_\Sigma \cup E_\varepsilon)$ defines a TNFA, $N = (V, \Sigma \cup \varepsilon, \delta, 0, \{n\})$.

Functionality $\mathcal{F}_{\mathsf{OT}}$

Define the security parameter κ.
- Upon receiving $(\mathsf{ot}, N, \ell, \{m_i\}_{i\in[0,n)})$ from P_0 and $(\mathsf{ot}, N, \ell, b)$ from P_1 such that $m_i \in \{0,1\}^\ell$ and $b \in [0, N)$, send m_b to P_1.
- Upon receiving (rot, N, ℓ) from P_0 and P_1, uniformly sample $(m_0, \ldots, m_{N-1}) \leftarrow \{0,1\}^{n\times\ell}$ and $b \in [0, N)$. Send $\{m_i\}_{i\in[0,N)}$ to P_0 and (b, m_b) to the P_1.

Figure 2: The functionality of oblivious transfer.

Functionality $\mathcal{F}_{\mathsf{2PC}}$

For $i \in [0, 1]$, upon receiving $(\mathcal{C}, x_i)$ from P_i, compute $(y_0, y_1) \leftarrow \mathcal{C}(x_0, x_1)$. Send y_i to P_i.

Figure 3: The functionality of two-party computation.

As the set of nodes in the directed graph is the set of states, we do not differentiate states and nodes without raising confusion in the rest of this paper. The output TNFA is equivalent to the language defined by re [SW11].

TNFA simulation. Taking as input TNFA G for re and a string $x = x_0 \cdots x_{m-1}$, the TNFA simulation decides if x is an element of $L(\mathsf{re})$ by checking whether or not G accepts x. The TNFA simulation algorithm is listed in Algorithm 1. It simulates the operation of an NFA by computing $\mathcal{Q}_i$, which includes all reachable states after processing the first i characters of x. The algorithm finally outputs if S_n is in $\mathcal{Q}_m$.

Computing $\mathcal{Q}_i$ from $\mathcal{Q}_{i-1}$ involves finding all states that can be accessed through a path $e_0, \cdots, e_\ell$, where $e_0 \in E_\Sigma$ is labeled x_{i-1} and $e_i \in E_\varepsilon$ for $i > 0$. The process is divided into two steps. First, the algorithm sets $\mathcal{Q}_i$ to all states that can be reached from $\mathcal{Q}_{i-1}$ through $e_0 \in E_\Sigma$ (lines 5-7 in Algorithm 1). As all edges in E_Σ have the form $(j-1, j)$, $\mathcal{Q}_i$ can be set up through a linear scan. This step is referred to as the *character transition*. Second, the algorithm extends $\mathcal{Q}_i$ to all states reachable from $\mathcal{Q}_i$ through any number of epsilon transitions (line 8 in Algorithm 1). This is done by finding all reachable nodes through a depth-first search (DFS). This step is referred to as the *epsilon transition*.

For simplicity of presentation of our protocols, from here onward we will represent a regular expression of size n by a graph of epsilon transitions $G = (V, E_\varepsilon)$ and an array re of size $n+1$, such that $\mathsf{re}[j]$ equals the character transition for the edge $(j, j+1)$ if it exists, else it is set to $\perp$.

2.2 Cryptographic Preliminaries

We define the following cryptographic building blocks and defer details of the protocol instantiation to Section 6.1.

Oblivious transfer. Oblivious transfer (OT) is a fundamental primitive for secure computation [Rab05]. It takes N messages $(m_0, \ldots, m_{N-1})$ from a sender and a choice index b from a receiver. The receiver receives m_b without knowing $\{m_i\}_{i\neq b}$ while the sender has no information on b. The random oblivious transfer (ROT) differs from OT by letting the functionality sample uniform messages for the sender and an index for the receiver. The functionalities are formally stated in Figure 2.

Two-party computation. Two-party computation (2PC) allows two mutually untrusted parties to jointly compute a public function over their private inputs without leaking anything beyond the output. We rely on the *Universal Composability* (UC) framework [Can20] to prove our two-party protocols are secure against semi-honest adversaries. The functionality is shown in Figure 3.

Garbled circuit. We employ Yao's garbled circuits [Yao82] (GC) to securely realize the functionality $\mathcal{F}_{\mathsf{2PC}}$.

Algorithm 1: TNFA simulation in plain text

```
   Input: An TNFA defined by $G(V, E)$ and string $x = x_0 \cdots x_{m-1}$
   Output: True/False
 1 Initiate a set $\mathcal{Q}_0 = \{S_0\}$
 2 Add all nodes that is reachable by $S_0$ in $G$ to $\mathcal{Q}_0$
 3 for $i = 0$ to $m - 1$ do
 4     Initiate a empty set $\mathcal{Q}_{i+1} = \phi$
 5     for $j = 1$ to $n$ do
 6         if $S_{j-1} \in \mathcal{Q}_i$ and $(j-1, j) \in E_\Sigma$ is labeled by $x_{i-1}$ then
 7             Put $S_j$ to $\mathcal{Q}_{i+1}$
 8                                                          ▷ Character transition
 9     $\mathcal{Q}_{i+1}$ ←DFS ($G_\varepsilon = (V, E_\varepsilon), \mathcal{Q}_{i+1}$)         ▷ Epsilon transition
10 return True if $S_n \in \mathcal{Q}_m$, and False otherwise.
```

The function to evaluate is represented as a Boolean circuit consisting of XOR and AND gates. A garbler P_0 constructs garbled tables that represent the wire values by random labels. An evaluator P_1 receives the input labels of the circuit and decrypts the output labels of each gate following the topological order. They derive the output by having the garbler interpret the output labels of the circuit.

Secret shares and their conversion. A binary value $x \in \{0, 1\}$ can be additively shared by two parties such that the sum of shares are equal to x. We consider mixed secret sharing schemes: Yao's garbled circuits and the additive secret shares [GMW19], as well as a practical approach to convert between these two types of sharings [DSZ].
- Yao's share ($[b]^Y$): P_0 holds a map $[b]^Y_0 := \{k_0 : 0; k_1 : 1\}$ and P_1 holds the key $[b]^Y_1 := k_b$.
- Additive share ($[b]^B$): P_0 holds $[b]^B_0$ and P_1 holds $[b]^B_1$ such that $[b]^B_0 \oplus [b]^B_1 = b$.
- Y2B ($[b]^Y \to [b]^B$): the conversion from Yao's shares to additive shares (local operation [DSZ]).
- B2Y ($[b]^B \to [b]^Y$): the conversion from additive shares to Yao's shares.

Oblivious stack. An oblivious stack data structure can be realized by the garbled circuits protocol. It allows conditional push and pop that take a secret Boolean value dictating whether the operation should be performed or disguised by a dummy execution [WNL+14]. We rely on the following operations.
- $\mathsf{ObStack} \leftarrow \mathsf{stack}()$: initialize a stack and a tag $\perp$;
- $(\cdot) \leftarrow \mathsf{ObStack.CondPush}([b], [x])$: push the element x to the oblivious stack if $b = 1$, else skip.
- $[x] \leftarrow \mathsf{ObStack.CondPop}([b])$: pop and return the top element x if $b = 1$, otherwise return $\perp$.

Zero-knowledge proofs. For a relation $\mathcal{R}$, public statement x, and private witness w, zero-knowledge proof of knowledge (ZKPoK) allows a prover $\mathcal{P}$ who proves $(w, x) \in \mathcal{R}$ to a verifier $\mathcal{V}$ without revealing any information of w. It enables $\mathcal{P}$ to validate a statement without compromising her privacy.

In terms of its security requirements: *Completeness* states that a valid witness always makes $\mathcal{V}$ accept. *Proof of knowledge* requires that, if $\mathcal{V}$ accepts, there is overwhelming probability that $\mathcal{P}$ holds a valid witness. *Zero-knowledge* means that no information related to the witness is leaked to $\mathcal{V}$ during the proving phase.

3 TNFA Simulation via Two Linear Scans

In this section, we describe our new TNFA simulation algorithm that is oblivious to the input string. The classical TNFA simulation algorithm 1 has an input-dependent memory access pattern and therefore is not suitable for private evaluation in circuit-based 2PC or ZKP. The input of the TNFA simulation algorithm

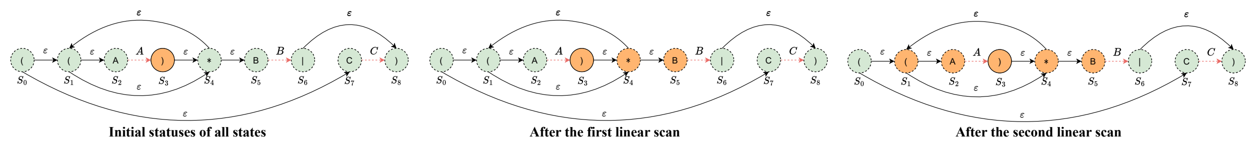
Initial statuses of all states
After the first linear scan
After the second linear scan

Figure 4: This figure demonstrates the activation of reachable states from a currently active state S_3. During the first linear scan, states S_1 and S_2 are not activated as they are reachable from state S_3 through a backward edge between S_4 and S_1. At the end of the first linear scan, state S_4 is activated and causes states S_1 and S_2 to become activated in the second linear scan.

includes the regular expression and the string. The privacy of either the input string or both the string and the regular expression is required. We first make the algorithm's access pattern independent of the input string by implementing the epsilon transition via linear scans. In Section 4, we explain how to utilize this linear scan-based simulation algorithm to perform regular expression matching in the ZKP setting. Later we show how to protect the privacy of both inputs via the oblivious stack or oblivious transfer in the 2PC scenario in Section 5.

The source of input dependence of the access pattern of the classical NFA simulation algorithm is the epsilon transition. In the epsilon transition, a depth-first search (DFS) over $G(V, E_\varepsilon)$ is invoked to add all reachable nodes from a set of nodes. The DFS algorithm results in memory access patterns dependent on the set of active states and the input graph.

We found that the DFS operation in the epsilon transition can be replaced by two linear scans. In algorithm 2, we represent the states of nodes using a bit vector s of length $n+1$. The vector is initialized to $s = (1, 0, \ldots, 0)$. A node S_i is said active if $s_i = 1$. The epsilon transition updates the ith bit from 0 to 1 if node S_i is reachable from a node S_j in the graph $G(V, E_\varepsilon)$ of the TNFA, where $s_j = 1$. To check reachability, it's sufficient to look at the states of a node's predecessors in the linear scan (line 12, Algorithm 2), as paths from u to v contain at most one backward edge if v is reachable from u. Nodes reachable by paths with only forward edges, such as S_5, are activated in the first linear scan, as their predecessors are activated first. Nodes reachable by paths with backward edges are activated in the second scan, as at least one of their predecessors is activated in the first scan. For example, in Figure 4, S_4 is activated in the first scan, making S_2 active in the second scan. Theorem 3.1 shows the correctness and the proof is deferred to Appendix A.2.

Definition 1 *An edge $(u_0, u_1) \in E_\varepsilon$ is deemed a backward edge if $u_1 < u_0$, otherwise it is a forward edge.*

Definition 2 *A pair of edges $(u_0, u_1), (v_0, v_1) \in E_\varepsilon$ are called **nested edges**, if $u_0 < v_0 < v_1 < u_1$ or $v_0 < u_0 < u_1 < v_1$, otherwise they are called a pair of **cross edges**.*

Lemma 3.1 *For $u, v \in V$, if v is reachable from u in $G(V, E_\varepsilon)$, then there exists a path from u to v that contains a maximum of one backward edge.*

Theorem 3.1 *Algorithm 2 is correct, i.e., for any $x \in \Sigma^*$ and TNFA N, the Algorithm 2 outputs $N(x)$.*

We show the proof of the above lemmas and theorems in Appendix A.

Optimization for wildcard. The wildcard metacharacter "?" can be used to represent any single character in a string, making it a powerful tool for pattern matching. For example, the wildcard pattern "a?b" would match any string that starts with "a" and ends with "b" with one letter in between, such as "acb" and "a1b". Classical TNFA simulation can perform pattern matching with wildcards using union operations for a finite alphabet, but this can be inefficient when the alphabet is large. Algorithm 2 avoids this issue by treating the wildcard metacharacter as a character that matches any character in character transition (See Line 10 in Algorithm 2) so that the cost of wildcard matching is independent of the size of the alphabet.

Complexity. Each invocation of the epsilon transition requires scanning nodes sequentially and updating their active bit twice. This depends on whether any of its predecessors are active and each node has at

Algorithm 2: TNFA simulation via two linear scans

```
Function 2ScanTNFAEval($x, N$):
    $s \leftarrow (1, 0, \ldots, 0) \in \{0,1\}^{n+1}$
    EpsilonTransitions($G(V, E_\varepsilon), s$)
    for $i = 0$ to $(m-1)$ do
        CharacterTransitions($x_i, \mathsf{re}, s$)
        EpsilonTransitions($G(V, E_\varepsilon), s$)
    return $s_n$
Function CharacterTransitions($x_i, \mathsf{re}, s$):
    for $j = n$ to $1$ do
        $s_j \leftarrow (s_{j-1} \wedge (\mathsf{re}_j \overset{?}{=} x_i)) \vee (\mathsf{re}_j \overset{?}{=} '?')$
    $s_0 \leftarrow 0$
Function EpsilonTransitions($G(V, E_\varepsilon), s$):
    for $t = 0$ to $1$ do
        for $j = 0$ to $n$ do
            $s_j \leftarrow s_j \vee (\vee_{(\alpha,j) \in E_\varepsilon} s_\alpha)$
```

maximum 2 predecessors (Lemma 5.1). Thus, the epsilon transition requires $O(n)$ time, which makes the TNFA evaluation run in $O(nm)$ time in total since it invokes the epsilon transition procedure m times.

4 ZK Regular Expression Matching

In this section, we introduce the Zero-knowledge Regular Expression (ZK-Regex) matching protocol. This protocol is designed for scenarios where the pattern for the regular expression is publicly known. The goal for the string holder (prover) is to prove to another party (verifier) that its private string matches or does not match the public regular expression while keeping the string confidential.

The ZK-Regex protocol leverages the TNFA simulation algorithm presented in Section 3 in a zero-knowledge proof setting. We also present a practical application of the ZK-Regex protocol, where a packet sender can prove to a network middlebox that an encrypted TLS message complies with a public regular expression without revealing its content. The network middleboxes such as firewalls or deep packet inspection devices could use this technique to enforce regulations while preserving the privacy of packet owners. This setting is relevant to systems like ZKMB [GAZ$^+$21] and DECO [ZMM$^+$20].

4.1 Standalone ZK-Regex

Our ZK-Regex scheme can be instantiated by any generic ZKP that supports proving the correctness of a statement that is translated from Algorithm 2. Assume a circuit $\mathcal{C}_{\mathsf{tnfa}}$ that is equivalent to Algorithm 2. It takes a private input the string $\mathbf{x}$ and a public input $(G(V, E_\epsilon), \mathsf{re})$, and outputs the last state of TNFA $s_n \in \{0,1\}$ which indicates whether the statement is false or true. The circuit $\mathcal{C}_{\mathsf{tnfa}}$ includes m invocations of character transitions and $m+1$ invocations of epsilon transitions. Each character transition requires $2n$ equality checks, with each check requiring $\log|\Sigma| - 1$ logical AND gates. Additionally, $2n$ extra logical AND gates are needed to combine the results. The epsilon transition involves $2 \times (n+1)$ iterations, each using at most 3 logical OR gates, based on the fact that each state has a maximum of two predecessors, as stated in Section 3. The total circuit for the zero-knowledge TNFA simulation requires $2mn \log|\Sigma| + 6(m+1)(n+1)$ logical AND gates for an input string of length m and a regular expression of length n.

We detail a ZK version of Algorithm 2 in Appendix B. In Section 6.2, we instantiate the ZK-regex from Limbo [DdSGOT21] and evaluate its performance.

4.2 ZK-Regex over TLS

To show it is practical in real-world scenarios, we apply our ZK-regex protocol to policy checking of TLS messages. Checking if an encrypted TLS message (e.g., encrypted DNS queries like DNS-over-TLS) complies with policies poses challenges because we want to protect user privacy even in places like K12 education where policy checking is mandatory to protect minors[McC04]. The use of ZK-Regex offers the best of both worlds: it ensures policy compliance while also preserving privacy.

For ZK-regex over TLS, we adopt the setting of [GAZ+21]: a verifier serves as a middlebox and publishes a set of regular expressions as policies, which defines legal contents. The prover, as a TLS packet sender, proves to the verifier whether the packet content is matched by regular expressions without decrypting it. To do so, the prover needs to show in the ZKP that 1) a private string is the decryption of a TLS ciphertext, and 2) whether the public regular expression matches the private string. A ZK proof with these two parts allows a network middlebox to inspect TLS-encrypted payloads without decrypting the packet. The ZK-regex described in Section 4.1 is able to prove the second task, however, the first task requires another ingredient that is referred to as the *Channel Opening* [GAZ+21].

Intuitively, channel opening can be done by letting the prover decrypt a TLS ciphertext in ZKP, which takes its encryption key as the witness. Unfortunately, this naive solution is not secure as observed by [GAZ+21; ZMM+20]. The reason is that the authenticated encryption schemes in TLS can be non-committing. Thus, the prover could use two different encryption keys EK_1,EK_2 to decrypt a single ciphertext $\mathbf{c}$ into two different messages M_1, M_2 where M_1 matches the regex while M_2 does not. Therefore, a malicious prover can generate a proof using M_1 and send M_2 through TLS (or vice versa). To solve this problem, the prover needs to prove that the key it uses to perform decryption in ZKP is the same key used for TLS sessions. The solution for the channel opening is forcing the prover to reiterate some parts of the TLS handshake in ZKP to re-derive the key, which binds the key with the handshake messages that can be collected by the verifier. We abstract three components of the circuit proven in ZKP. The first two are generalized from ZKMB [GAZ+21].

- $\mathsf{TLS.Handshake}(\mathsf{HSSec}) \to \mathsf{EK}$. It takes input partial handshake secrets HSSec in TLS handshake and outputs an encryption key EK for the TLS record layer.
- $\mathsf{TLS.RecordDec}(\mathbf{c}, \mathsf{EK}) \to \mathbf{x}$. It takes input a TLS ciphertext $\mathbf{c}$ and the encryption key EK, and outputs a string $\mathbf{x}$.
- $\mathsf{PatternMatch}(\mathsf{re}, \mathbf{x}) \to \{0, 1\}$. It takes input a regex re and the string $\mathbf{x}$, and outputs whether $\mathbf{x}$ is matched by re.

Assume the verifier inspects the network communication and records all TLS messages transited between the packet sender and receiver. At the first step, the prover (packet sender) takes input the secrets HSSec derived from Diffie-Hellman key exchange during the TLS handshake, and proves that EK is indeed the TLS encryption and authentication key. Next, the prover extracts the encrypted messages $\mathbf{x}$ in ZKP, which is the private input to ZK-regex for the proof of pattern match. We define the ZK-Regex-over-TLS as follows.

Definition 3 *The functionality of ZK-Regex-over-TLS takes private input* $\mathbf{x}$*, handshake secrets* HSSec*, and public TLS ciphertext* $\mathbf{c}$*. Let both* re *and* Σ *be public. It outputs* accept *to* $\mathcal{V}$ *if* $\mathbf{x} \in L(\mathsf{re})$ *and* $\mathbf{x} = \mathsf{TLS.RecordDec}(\mathbf{c}, \mathsf{TLS.Handshake}(\mathsf{HSSec}))$*. Otherwise, it outputs* reject*.*

Our innovation is mainly on $\mathsf{PatternMatch}$ thus we refer to [GAZ+21] (Appendix B, C) for the detailed protocols that efficiently realize $\mathsf{TLS.Handshake}$ and $\mathsf{TLS.RecordDec}$, along with their security proofs. We provide more discussion on the instantiation of Channel Opening in Section 6.1. In Section 6.2, we evaluate an end-to-end implementation of the above application and show its performance on DNS filters from the Pi-hole project [pi-22].

5 Secure Two-Party Regular Expression Matching

In many practical situations, both the regular expression and input string must be kept private. In this section, we introduce our protocols for secure two-party regular expression matching (secure-regex). Based on the TNFA simulation detailed in Algorithm 2, secure-regex consists of two components: character transition and epsilon transition. The character transition involves comparing each character of a length-n regular expression to a single character of the input string and updating the states based on the comparison results. This process is represented by n comparison circuits of size $\log|\Sigma| - 1$ and $2n$ logical AND gates for state updates. These character transitions can be easily expressed as a Boolean circuit and securely evaluated through 2PC. The communication complexity for privately performing all character transitions in a pattern matching task is $O(\kappa mn \log|\Sigma|)$ when 2PC is implemented through the Garbled Circuit (GC) method.

Implementing epsilon transitions in Algorithm 2 directly through GC is difficult due to the need to keep the interconnection of the graph $G(V, E_\varepsilon)$ secret from the string holder. The state of a node in the graph is propagated from its predecessors, and the 2PC protocol must perform an oblivious retrieval of a state from a list of secretly shared states. This type of oblivious table lookup operation is often challenging. While using oblivious RAM could solve the problem generically, it would bring significant overhead and make the protocol inefficient [GO96].

The two methods for efficient handling of the epsilon transition are presented in the subsequent sections: one based on the oblivious stack (OS) and another based on the oblivious transfer (OT). The OT-based approach incurs lower communication overhead for short input regular expressions, but its cost increases faster than OS-based protocol, and thus is less efficient for long regular expressions. Also, the OS-based approach incurs only a constant number of round-trip communication and has a significant advantage in the high-latency network. We defer our detailed performance evaluation and comparison to Section 6.

5.1 Epsilon Transition via Oblivious Stack

This section presents a constant-round two-party computation protocol that employs an oblivious stack for epsilon transitions. The protocol requires $O(n)$ stack operations with an average communication complexity of $O(\log n)$ per operation. First, we introduce the following definitions and lemmas that specify these patterns.

Definition 4 *An edge $(u_0, u_1) \in E_\varepsilon$ is termed a **long edge** if $|u_1 - u_0| > 1$. In addition, (u_0, u_1) is called u_0's outgoing edge or u_1's incoming edge.*

Lemma 5.1 *Every node in the TNFA graph G has at most one incoming long-forward edge and at most one outgoing long-forward edge in E_ε.*

Lemma 5.2 *Every pair of backward edges is nested.*

Lemma 5.3 *A pair of long-forward edges (u_0, u_1) and (v_0, v_1) with $v_0 < u_0$ are cross edges if and only if $v_1 = u_0 + 1$ and $re[u_0] =' |'$.*

Lemma 5.2 and Lemma 5.3 illustrate that all backward edges are pairwise nested, and forward edges are almost nested with the exception of the scenario described in Lemma 5.3. We show the proof of the above lemmas in Appendix A.2.

Oblivious graph representation. By Lemma 5.1, the number of incoming and outgoing edges for each node in $G = (V, E_\varepsilon)$ is limited to 2. In addition, each node has at most one incoming and outgoing long edge. Therefore, G can be represented by six $(n + 1)$-bit binary vectors $\mathsf{hasSFIE}$, $\mathsf{hasLFIE}$, hasBIE, $\mathsf{hasLFOE}$, hasBOE, isOR, in which the i-th bit representing whether the i-th node has short forward incoming/ long forward incoming/backward incoming/long forward outgoing/backward outgoing edges and whether the i-th character in regex is $'|'$.

With the oblivious graph representation of the TNFA, we present the formal outline of our oblivious stack-based circuit in Algorithm 5. The algorithm consists of two subprotocols: the *forward scan* (Algorithm 3) and the *backward scan* (Algorithm 4).

Algorithm 3: ForwardScan

```
1  Input: $G = (V, E_\varepsilon)$ presented by hasSFIE, hasLFIE, hasLFOE, isOR and states $\{s_0 \ldots, s_n\}$
2  ostack ← stack()
3  cross ← false, tmp ← false
4  for $j = 1$ to $n$ do
5      popElmt ← ostack.pop((hasLFIE[j] ∧ ¬cross) ∨ isOR[j])
6      $s_\alpha \leftarrow \mathbf{Ite}(\mathsf{cross}, \mathsf{tmp}, \mathsf{popElmt})$ // If-then-else
7      $s_\alpha \leftarrow \mathbf{Ite}(\mathsf{hasLFIE}[j], s_\alpha, \mathsf{false})$
8      $s'_\alpha \leftarrow \mathbf{Ite}(\mathsf{hasSFIE}[j], s_{j-1}, \mathsf{false})$
9      $s_j \leftarrow s_j \lor s_\alpha \lor s_{\alpha'}$
10     ostack.push(hasLFOE[$j$], $s_j$)
11     cross ← isOR[$j$]
12     tmp ← $\mathbf{Ite}$(cross, popElmt, false)
```

Algorithm 4: BackwardScan

```
1  Input: $G = (V, E_\varepsilon)$ presented by hasBIE, hasBOE and states $\{s_0 \ldots, s_n\}$
2  ostack ← stack()
3  for $j = n$ to 0 do
4      $\alpha \leftarrow 0$
5      $\alpha \leftarrow$ ostack.pop(hasBIE[$j$])
6      $[s_j] \leftarrow s_j \lor \alpha$
7      ostack.push(hasBOE[$j$], $s_j$)
```

In the backward scan subprotocol, nodes are processed from n to 0 and activated (by setting their s entry to true) if they have an incoming backward edge from an active node. When reaching an outgoing backward edge from state j, s_j is pushed into the stack. When the current state s_j has an incoming backward edge, a state bit is popped from the stack and its status is propagated to s_j. Due to the nested property and stack data structure, the backward scan ensures correctness, meaning any node reachable from an active node via a backward edge is activated.

The objective of the forward scan is to activate a node if there is a path to it from an active node that consists only of forward edges. Table 2 provides an illustration of the forward scan subprotocol. Unlike backward edges, forward edges can cross, as shown by the crossing of S_6 and S_7 in Figure 1. However, according to Lemma 5.3, these crossings can only occur between adjacent nodes and can be identified by examining if the character is $'|'$ in re.

To handle these special crossings, the stack-based approach used for backward edges can be adjusted. When the stack contains two nodes with crossed forward edges, such as s_0, s_6 in the processing of S_6, the top two elements can be swapped to s_6, s_0. This allows s_0 to be popped from the stack and used for updating the next state with a long forward incoming edge (LFIE), S_7 in this case. However, swapping the elements causes three operations on the stack. To reduce the number of operations, the variable tmp is introduced to store the top bit of the stack instead of swapping. After processing S_6, the stack will have s_6 and $\mathsf{tmp} = s_0$,

State	stack	s_α	s'_α	cross	tmp	Long Edge
S_0	s_0	F	F	F	F	Has LFOE
S_1	s_0, s_1	F	s_0	F	F	Has LFOE
S_2	s_0, s_1	F	s_1	F	F	None
S_3	s_0, s_1	F	F	F	F	None
S_4	s_0	s_1	s_3	F	F	Has LIFE
S_5	s_0	F	s_4	F	F	Has LIFE
S_6	s_6	F	F	T	s_0	Has LFOE
S_7	s_7	s_0	F	F	F	Has LFIE
S_8		s_6	F	F	F	Has LFIE

Table 2: Example of one forward scan for the TNFA in Algorithm 3. The ith row in the table shows the status of the stack and variable values after processing the ith state. If a state has a long forward outgoing edge (LFOE), its status is always pushed onto the stack. If it has a long forward incoming edge (LFIE), its status is updated using either a bit popped from the stack or stored in tmp. For example, S_4 in Figure 1 has a LFIE caused by $'*'$, so S_α is assigned the bit popped from the stack. Meanwhile, S_7 has a LFIE caused by $'|'$, so tmp stores S_0 after processing S_6 and is used to update S_7.

Algorithm 5: Epsilon Transition via Oblivious Stack

```
1 Input: $N = (G = (V, E_\varepsilon), \mathsf{re})$ and states $\{s_0, \ldots, s_n\}$ that are shared by two parties.
2 ForwardScan($G = (V, E_\varepsilon), \{s_0, \ldots, s_n\}$)
3 BackwardScan($G = (V, E_\varepsilon), \{s_0, \ldots, s_n\}$)
4 ForwardScan($G = (V, E_\varepsilon), \{s_0, \ldots, s_n\}$)
  Output: Updated shared states' statuses $\{s_0, \ldots, s_n\}$
```

which will be used immediately for the next state with an LFIE. This approach is valid due to the structure of the TNFA presented in Lemma 5.3.

Security. All components in the Algorithms 3, Algorithm4, and Algorithm 5, are realized in $\mathcal{F}_{\mathsf{2PC}}$. Thus the security follows the instantiation of the underlying generic two-party computation. We discuss more about the realization of the protocol components in Section 6.1. We provide the following theorem and defer the proof to Appendix C.4.

Theorem 5.1 *The protocol shown in Algorithm 5 securely realizes the function* EpsilonTransition *described in Algorithm 2 (Lines 12-15) against semi-honest adversaries in the $\mathcal{F}_{\mathsf{2PC}}$-hybrid mode.*

Complexity. An epsilon transition requires a total of $6n + 2$ stack accesses. The efficient construction of oblivious stacks [ZE13] for a stack of size $O(n)$ incurs $O(\log n)$ amortized cost for each oblivious access. Thus the net circuit size for oblivious stack is $O(n \log n)$. Additionally, both forward and backward scans take $O(n)$ AND gates for non-stack operations. Hence, the epsilon transition has $O(n \log n)$ circuit complexity (in the number of AND gates). If the epsilon transition is realized by this stack-based protocol and the functionality $\mathcal{F}_{\mathsf{2PC}}$ is instantiated by the garbled circuits protocol, the communication complexity of the regular expression matching task (shown in Algorithm 2) is $O(\kappa m n (\log n + \log |\Sigma|))$.

5.2 Epsilon Transition via 1-out-n+1 OT

An important observation that helps optimize the OS-based protocol is that the predecessors of each state in a TNFA are solely determined by the input regular expression. As a result, the pattern holder knows the

```
Algorithm 6: Epsilon Transition via 1-out-of-n+1 OTs
Input: String holder inputs Boolean shares of states' statuses $\{[s_0]_0^B, \ldots, [s_n]_0^B, \}$
Input: Pattern holder inputs $G(V, E_\varepsilon)$, and her Boolean shares of states' statuses $\{[s_0]_1^B, \ldots [s_n]_1^B\}$
1 for $t = 0$ to 1; $j = 0$ to $n$ do
2     String holder samples a random bit $r_b$, and sends $(\mathsf{ot}, n+1, 1, \{[s_j]_0^B \oplus r_b\}_{j\in[0,n]})$ to $\mathcal{F}_{\mathsf{OT}}$
3     Denote $\alpha$ as the index of $j$-th state's predecessor. Pattern holder sends $(\mathsf{ot}, n+1, 1, \alpha)$ to $\mathcal{F}_{\mathsf{OT}}$
4     $\mathcal{F}_{\mathsf{OT}}$ returns $m = [s_\alpha]_0^B \oplus r_b$ to the pattern holder
5     Pattern holder sets her new share $[s_\alpha]_1^B \leftarrow m \oplus [s_\alpha]_1^B$
6     String holder sets her new share $[s_\alpha]_0^B \leftarrow r_b$
7     For another predecessor of $j$-th state $\alpha'$, two parties repeat the above to get $[s_{\alpha'}]_0^B$ and $[s_{\alpha'}]_1^B$
        respectively
8     Invoke $\mathcal{F}_{\mathsf{2pc}}$ with B2Y to compute
        $[s_j]_0^Y, [s_j]_1^Y \leftarrow ([s_j]_0^B \oplus [s_j]_1^B) \vee ([s_\alpha]_0^B \oplus [s_\alpha]_1^B) \vee ([s'_\alpha]_0^B \oplus [s'_\alpha]_1^B)$
9     Translate Yao's share to additive share as $[s_j]_0^B, [s_j]_1^B \leftarrow \mathrm{Y2B}([s_j]_0^Y, [s_j]_1^Y)$
Output: String holder holds shares of updated states $\{[s_0]_0^B, \ldots, [s_n]_0^B\}$
Output: Pattern holder holds shares of updated states $\{[s_0]_1^B, \ldots, [s_n]_1^B\}$
```

exact states (i.e., the value of α at line 15 in Algorithm 2) that need to be retrieved when updating a state during epsilon transitions. Based on this, we propose an alternative OT-based approach for secure epsilon transition that has a total communication overhead of $O(mn(n+\kappa\log|\Sigma|))$. While its asymptotic overhead is not as good as the OS-based approach, it is still practical for matching with short regular expressions or in low-latency networks. A comprehensive comparison with the OS-based approach will be provided in Section 6.

The protocol utilizes the functionalities $\mathcal{F}_{\mathsf{2PC}}$, $\mathcal{F}_{\mathsf{OT}}$, and the conversion functions Y2B and B2Y discussed in Section 2.2. It is based on the bound on the number of predecessors for each state established in Lemma 5.1, which states that a state can have at most 2 predecessors. The protocol starts by having the two parties secretly share the states $(s_0, \ldots, s_n) \in \{0,1\}^{n+1}$. The goal is to update a state s_j based on its two predecessors s_α and $s_{\alpha'}$ if they exist. To do this, the parties use 1-out-of-n+1 OT twice to retrieve the secret shares of α-th and α'-th state (lines 2-7). Then they employ $\mathcal{F}_{\mathsf{2PC}}$ to update the secret shared state s_j (line 8). Y2B and B2Y are utilized to convert between the forms of the secret shares when transforming between $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{2PC}}$ (lines 8-9).

The security of the protocol in Algorithm 6 is stated below and its formal proof is provided in Appendix C.3.

Theorem 5.2 *The protocol shown in Algorithm 6 securely realizes the function* EpsilonTransition *described in Algorithm 2 (Lines 12-15) against semi-honest adversaries in the* $(\mathcal{F}_{\mathsf{2PC}}, \mathcal{F}_{\mathsf{OT}})$*-hybrid mode.*

Complexity. Algorithm 6 makes $4(n+1)$ invocations of 1-out-of-n+1 OT. The corresponding garbled circuit(GC) contains $4(n+1)$ inputs from both the garbler and evaluator and $4(n+1)$ AND gates. The $4(n+1)$ invocations of 1-out-of-n+1 OT only require the string holder to send $4(n+1)^2$ bits to the pattern holder during the online phase. We will detail how we achieve $(n+1)$-bit per 1-out-of-n+1 OT in the online phase with a sublinear preprocessing in Section 6.1. Assume that the string holder is the garbler and the pattern holder is the evaluator of GC. Among $2(n+1)$ rounds, the garbler sends 4κ bits and the evaluator sends 2 bits per round. This epsilon transition protocol incurs a total of $4(n+1)^2 + 8(n+1)\kappa + 4(n+1)$ bits communication overhead and $2(n+1)$ round trips. When applying this OT-based epsilon transition to the TNFA simulation described in Algorithm 2, it costs $O(mn(n+\kappa\log|\Sigma|))$ communication and $O(mn)$ round-trips.

Algorithm 7: 1-out-of-N OT

```
1 Initialization ($\alpha$):
2     $P_0$ and $P_1$ send $(\mathsf{rot}, N, 1)$ to $\mathcal{F}_{\mathsf{OT}}$, which returns $(r_0, \cdots, r_{N-1}) \in \{0,1\}^N$ to $P_0$ and
      $(\gamma, r_\gamma) \in \mathbb{N} \times \{0,1\}$ to $P_1$
3     Receiver sends $\delta = \alpha - \gamma \mod N$ to $P_0$.
4 Online $(x_0, \ldots, x_{N-1}) \in \{0,1\}^N$:
5     $P_0$ computes and sends $\mathbf{y} = (x_0 \oplus r_{i_0}, \cdots, x_{N-1} \oplus r_{i_{N-1}})$ to $P_1$, in which $i_k = k - \delta$
      $\mod N$ for $k \in [0, N)$.
6     Receiver computes the output $x_\alpha = y_\alpha \oplus r_\gamma$.
```

6 Performance Evaluation

We first describe the detailed instantiation of cryptographic protocols we described in our protocol. Then we provide concrete performance evaluations of our privacy-preserving pattern matching protocols. We evaluate both our secure-regex and zk-regex protocols with varying message and pattern lengths, and we test their performance with real-world tasks to show the practicality of our protocols.

6.1 Instantiate Cryptographic Building Blocks

1-out-of-N OT. Similar to the previous 1-out-of-N OT schemes, we instantiate it by converting from a 1-out-of-N ROT [NP99; KKRT16; KK13]. When plugging into Algorithm 6, we make use of the fact that the choice indices are determined by the pattern. Thus at the beginning of the protocol execution, the pattern holder is able to send all differences of the real choice indices and random choice indices obtained from ROT. It allows to efficiently generate 1-out-of-N ROT messages during a preprocessing phase and only incurs one-round communications during the online phase. The conversion is described in Algorithm 7. We define its security proof to Appendix C.1.

1-out-of-N ROT. We describe an efficient 1-out-of-N ROT protocol that realizes the corresponding functionality in $\mathcal{F}_{\mathsf{OT}}$ (Algorithm 2). The protocol takes input $\lceil \log_2 N \rceil$ 1-out-of-2 ROT messages and transforms them into 1-out-of-N ROT with $O(N)$ invocations of correlation robust hash (CRH) [GKWY20]. While instantiating the 1-out-of-2 ROT with the subfield vector oblivious linear evaluation (VOLE) based on pseudorandom correlation generator (PCG) [BCG$^+$19a; YWL$^+$20; BCG$^+$19b], the communication cost is less than $\lceil \log_2 N \rceil$ bit in total. We defer the detailed protocol description, optimization, and security proofs to Appendix C.2.

Garbled circuits. The garbled circuits that realizes $\mathcal{F}_{\mathsf{2PC}}$ is instantiated by the half-gates protocol [ZRE15], with optimizations including free-XOR [KS08], point-and-permute [BMR90] and garbled row reduction [NPS99]. Assuming the sublinear ROT during the preprocessing, the online communication includes 1 bit per evaluator's input and 2κ bit per AND gate.

Oblivious stack. We use the oblivious stack protocol proposed in [ZE13]. Each conditional push or conditional pop operation incurs $O(\kappa \log n)$ amortized communication complexity when instantiated in garbled circuits. At a high level, it builds a layered data structure with increased capacity from the lower layers to the upper layers. The pushed elements are first stored at lower layers. All elements at one layer will be propagated to the upper layer when the current layer is possibly full. The pop operations reverse this process. We refer to [ZE13] for more details about the protocol. In our prototype, we use the implementation open-sourced by [LJA+22].

MPC-in-the-Head. We instantiate the MPCitH protocol by the Limbo framework [DdSGOT21]. Limbo adopts an MPC verification protocol that is based on additive secret sharing and a sublinear distributed multiplication triple checking scheme [BBCG+19; GSZ20]. After being compiled to a ZKP, it results in high efficiency, non-interactiveness, and proof size linear to the circuit size. It is currently the state-of-the-art MPCitH protocol. The only work in this regime that achieves sublinear proof size is Ligero [AHIV17]. However, its computation is more expensive than Limbo.

Shortcut Channel opening for TLS 1.3. Recall from Section 4.2 that channel opening refers to the circuit that proves a private string is the decryption of a public TLS ciphertext, and it must re-derive the session key to prevent key equivocation. A naive approach is to repeat all client key derivations from the TLS Handshake, but this is inefficient due to costly group operations and hashing of a potentially long transcript, and thus makes the circuit size dependent on the longest possible transcript. ZKMB [GAZ+21] offers a shortcut by only re-executing essential intermediate operations from the handshake process, eliminating those expensive operations. Our implementation takes this approach, resulting in a circuit that does not contain any group operations, and its size is independent of the transcript length. Specifically, we implemented the shortcut channel opening circuit in Figure 9 of [GAZ+21]. We refer to Appendix B and C of [GAZ+21] for a detailed description of the TLS channel opening protocol and related security proofs.

6.2 Performance Evaluation of ZK-Regex

We now evaluate the performance of our ZK-regex protocol. First, we measure the concrete prover time and proof size of our standalone ZK-regex protocol with varying m and n as well as a real-world example using the Pi-hole filters [pi-22]. Then, we deploy our protocol over TLS and show our ZK-regex over TLS is practical in a realistic DNS policy checking setting.

We implemented ZK-regex protocol and channel opening for TLS using emp-toolkit [WMK16] and evaluated our circuits in Limbo framework [DdSGOT21] with parameters that achieve 2^{-40} statistical security. Specifically, we fix the compression factor to 32 and the number of parties to 16 for all runs. All experiments in this section use a single Amazon EC2 `m5.4xlarge` instance with 64 gigabytes of RAM. We assume $|\Sigma| = 256$ so that both pattern length and string length are in bytes. We note that there is an inherent trade-off between proof size and prover time within MPCitH: increasing the number of parties in MPCitH will decrease the proof size but it will also increase prover time. This trade-off makes ZK-regex versatile and well-suited for a range of settings with different priorities. We choose these parameters in our experiments to reflect the practicality of our protocol in terms of both prover time and proof size.

r1	97	379KB	0.57s	r2	39	169KB	0.23s
r3	66	267KB	0.38s	r4	69	277KB	0.39s
r5	83	328KB	0.47s	r6	37	161KB	0.22s
r7	23	111KB	0.14s	r8	21	103KB	0.13s
r9	59	241KB	0.35s	r10	65	263KB	0.37s
r11	6	49KB	0.05s	r12	16	85KB	0.1s
r13	68	274KB	0.39s	Avg.	49	208KB	0.29s

Table 3: The Length of the regular expression, proof size and prover time of ZK-regex applied to Pi-hole regular expression set. The patterns (r1,...,r13) are provided in [pi-20]. We omitted one pattern because it cannot be efficiently translated.

Benchmark the protocol. First, we measure the proof size and prover time with varying input strings and

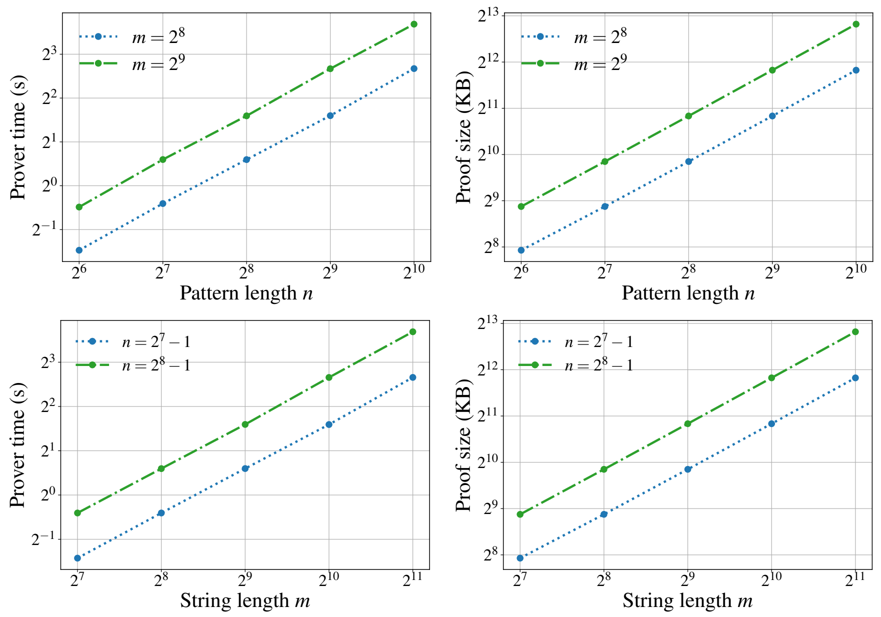

Figure 5: The prover time and proof size of our ZK-regex protocol (average over 10 executions)

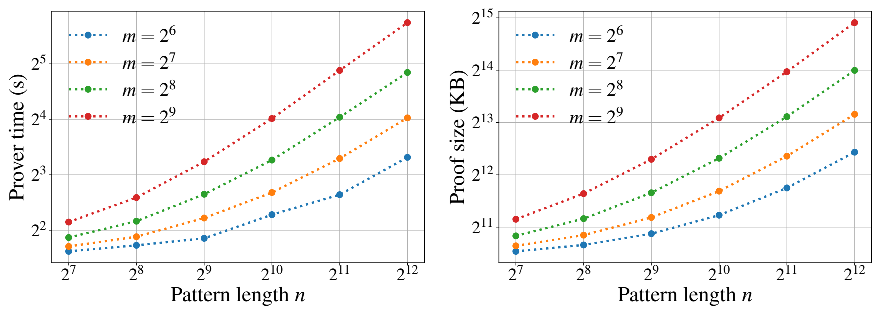

Figure 6: The prover time and proof size of our ZK-regex protocol deployed over TLS 1.3 (average over 10 executions)

regular expression length. The result appears in Figure 5. The proof size and prover time grow linearly with both the input string and regular expression length. For a 128-byte string and 128-byte regex (typical for real-world scenarios), our protocol generates a proof of 243.8 KB in 0.37 s.

Next, in our DNS policy checking example, we use the regular expressions for blocking filters provided by Pi-hole [pi-20]. The evaluation result appears in Table 3. For the longest pattern r1, ZK-regex generates a proof of size 379 KB in just 0.57 s.

Comparison with [FKL$^+$21]. The protocol proposed by Franzese et al. [FKL$^+$21] is based on a constant-overhead oblivious array access in the ZK setting and its pattern match relies on the DFA. The representation of DFA usually leads to an exponential explosion in terms of the length of the regular expression. It requires $O(n2^n|\Sigma|)$ to build the array to store the DFA and $O(mn^2)$ to traverse it. Although its read operation only takes 15μs per access, the efficiency and scalability of [FKL$^+$21] is not comparable to our ZK-regex.

Benchmark the ZK-regex over TLS. We implement our ZK-regex protocol over TLS by integrating the shortcut channel opening in our ZK-regex circuit. The cost of our proof now consists of two parts: shortcut channel opening and zk-regex. Although the addition of the channel opening subcircuit imposes some overhead, we show it is still practical by evaluating our ZK-regex over TLS in a realistic DNS policy checking setting. We evaluated the prover time and proof size of our ZK-regex deployed over TLS 1.3 with varying m and n. The result appears in Figure 6. When ZK-regex over TLS is applied to DNS policy checking using

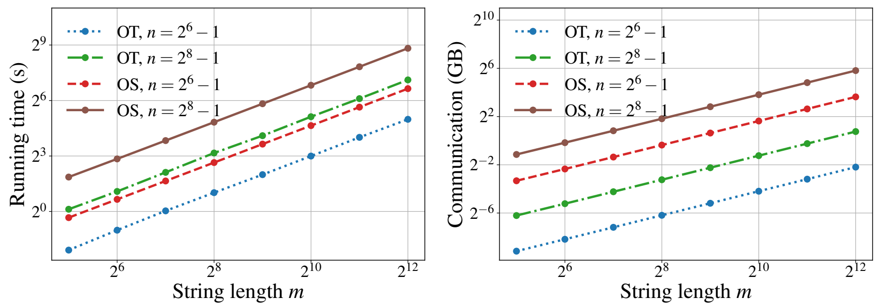

Figure 7: The running time (in seconds) and communication overhead (in gigabytes) of our OT-based and OS-based protocol with subject to the change of string length (m). The network bandwidth is fixed to 1 Gbps.

Bandwidth(Mbps)	50	100	500	1K	5K
OT	**18.2**	**9.1**	**8.9**	**8.8**	**8.4**
OS	568.4	284.2	56.8	28.4	11.8

Table 4: The running time (seconds) of the protocol with subject to the bandwidth. The pattern length $n = 255$ and the string length $m = 256$.

the Pi-hole pattern set, the longest pattern with $n = 97$ and $m = 128$ yielded a proof that took only 2.88 seconds to generate and had a size of just 1372 KB. We remark that the overhead imposed by the channel opening circuit is independent of n and only linear to m.

6.3 Performance Evaluation of Secure-Regex

We implement our privacy-preserving pattern match protocols and demonstrate their performance through a series of experiments. Our use of garbled circuits protocol is built on top of the emp-toolkit [WMK16]. We implement the 1-out-of-$n+1$ oblivious transfer (OT) protocol described in Section 6.1, and use the oblivious stack proposed in [ZE13] and open sourced from [LJA+22]. We use two Amazon EC2 `m5.xlarge` instances located in the same region to act as the string holder and the pattern holder. The instances are equipped with 4 vCPUs with clock speed up to 3.1 GHz, 16 GiB RAM and up to 10 Gbps network bandwidth. We use only a single thread. Meanwhile, we use the Linux traffic control tool `tc` to control the network bandwidth and create latency to simulate the wide area network (WAN).

Benchmark the protocols. We first fix the network bandwidth to 1 Gbps and show the performance with regard to variables n (pattern length) and m (string length). We report the running time and communication overhead for each execution for both approaches (OT-based or OS-based). Recall that asymptotically, the communication complexity of the OT-based approach is $O(mn(n + \kappa \log|\Sigma|))$ and the OS-based approach is $O(\kappa mn(\log n + \log|\Sigma|))$. Figure 7 demonstrates how the protocols scale with the increasing of the input string length m. We choose the parameters $(n = 2^i - 1, m = 2^j)$ where $i \in \{6, 8\}$ and $j \in [5, 12]$. Both the running time and the communication overhead for two protocols are linear to m, which aligns with our analysis. Figure 8 shows the performance with the increased pattern length n. We choose the parameters $(n = 2^i - 1, m = 2^j)$ where $i \in [5, 13]$ and $j \in \{6, 8\}$. For short patterns, the communication overhead of the OT-based approach is dominated by the character transition, thus is still linear to n. As n increases, the cost of epsilon transition dominates thus the overhead becomes quadratic to n. The performance of the OS-based approach complies with our analysis. Overall the OT-based approach is more efficient for short patterns, but is outperformed by OS-based when $\log n > 12$.

Next, we fix the pattern length $n = 255$ and string length $m = 256$. To demonstrate how our protocols react to the change in the network bandwidth, we report the running time under different network settings.

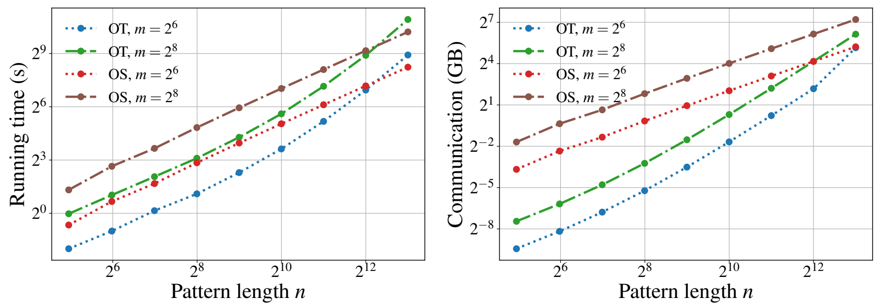

Figure 8: The running time (in seconds) and communication overhead (in gigabytes) of our OT-based and OS-based protocol subject to changes in pattern length (n). The network bandwidth is fixed to 1 Gbps.

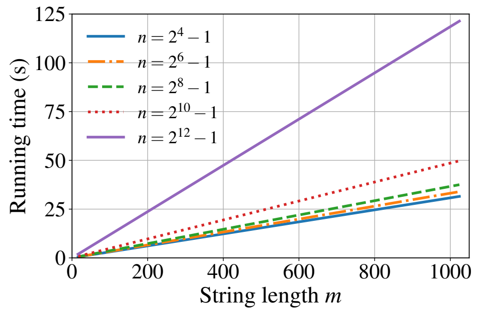

Figure 9: The running time (in seconds) of our OT-based protocol applied to regular expressions (RE) from the SNORT system. The network bandwidth is set to be 1 Gbps.

In Table 4, we report the running time of two protocols while increasing the network bandwidth. The performance of the OT-based protocol does not change when the bandwidth is higher than 100 Mbps due to its low communication overhead. The running time of the OS-based protocol decrease as the bandwidth is raised from 50 Mbps to 5000 Mbps, which shows that communication is its bottleneck.

The influence of the network latency on our protocol is demonstrated in Table 5. We fix the pattern length $n = 63$ and string length $m = 64$. We choose the maximum latency to be 60 milliseconds, which is the round-trip delay between the US west and east coast. This is a proper simulation of real-world WAN. The OT-based protocol has a number of round-trips linear to the pattern length n and string length m, thus it deteriorates quickly when the latency increases. The OS-based protocol is less affected by the latency due to its constant round of communication and performs better than the OT-based approach in all delayed network settings. Overall, the OT-based approach shows better performance in low-bandwidth, low-latency,

Scheme	0 ms	2 ms	20 ms	40 ms	60 ms
OT	**0.5**	17.2	167.3	333.75	500.2
OS	1.5	**1.6**	**1.7**	**2.8**	**14.2**

Table 5: The running time (seconds) of the protocol with subject to the network latency. The pattern length $n = 63$ and the string length is $m = 64$.

and short-pattern settings, while the OS-based protocol is more efficient in high-bandwidth, high-latency, and long-pattern settings.

Benchmarks against SNORT. We demonstrate the use of secure-regex in resolving privacy concerns in real-world intrusion detection. A pattern holder holds rule-based regular expressions and can identify potential threats from incoming network packets, while a string holder holds confidential network packets from clients. To protect both the rules and packet contents, the two parties can use secure-regex protocols to perform secure pattern matching.

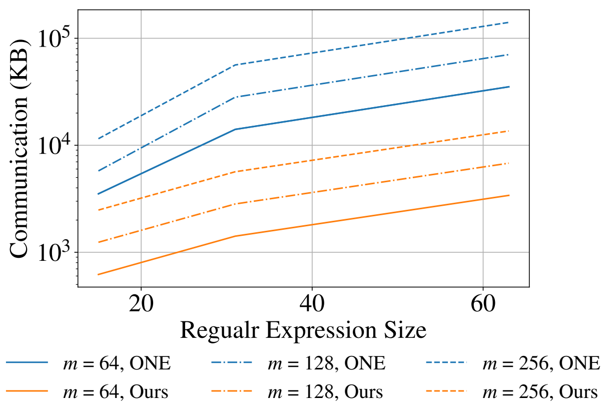

Figure 10: Comparisons with ONE. n is the size of the pattern The results show that our OT-based secure-regex improved communication cost by 4.5X.

We use the open-source SNORT PCRE (Perl Compatible Regular Expressions) to showcase the use of regular expressions in intrusion detection [CIS]. SNORT PCRE is a sophisticated form of regular expressions with advanced features like backreferences and recursive patterns. Our focus is on classical regular expressions, which typically support only disjunction, concatenation, and loop operations. We rewrite the SNORT PCRE into our format described in Section 2.1. This results in 416 SNORT PCRE that are suitable for use in our secure-regex protocol, with 356 (85.5%) of them having a length of $\leq 2^{12}$ bytes.

We benchmark our secure-regex protocol against regular expressions of SNORT, ignoring the regex that has length $> 2^{12}$. We show in Figure 9 the running time of secure-regex (OT-based approach) with partial SNORT PCRE against input string of varied length. Note that our secure-regex implementation requires the length of regular expressions padded to almost a power of 2. For clarity we only benchmark for $n = 2^j - 1, j \in \{4, 6, 8, 10, 12\}$.

Comparison with ONE Sasakawa et al. introduced ONE as a state-of-the-art. solution for regular expression pattern matching utilizing NFA in their paper [SHd+14]. Although the implementation of ONE is not presently accessible, we evaluated the communication cost of our OT-based protocols against Sasakawa et al.'s method, relying on the performance metrics they reported in their study. According to the findings, the utilization of our OT-based secure-regex brings communication cost reduction of more than 4.5X.

7 Acknowledgement

Work by Ning Luo, Gefei Tan, and Chenkai Weng was supported by NSF award 2236819 and Research awards from Google. Work by Ruzica Piskac was supported in part by NSF award CNS-1562888 and CCF-2131476.

References

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2087–2104, 2017.

[APD+10] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for dns. In *USENIX Security Symposium*, pages 273–290, 2010.

[BBCG$^+$19] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear pcps. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference*, pages 67–97. Springer, 2019.

[BCG$^+$19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 291–308, 2019.

[BCG$^+$19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent ot extension and more. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference*, pages 489–518. Springer, 2019.

[BEDM$^+$13] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 5pm: Secure pattern matching. *Journal of computer security*, 21(5):601–625, 2013.

[Bee09] Nicole Beebe. Digital forensic research: The good, the bad and the unaddressed. In *IFIP International conference on digital forensics*, pages 17–36. Springer, 2009.

[Ben99] Gary Benson. Tandem repeats finder: a program to analyze dna sequences. *Nucleic acids research*, 27(2):573–580, 1999.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 310–321, 2002.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513, 1990.

[BSA13] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 207–218, 2013.

[Can20] Ran Canetti. Universally composable security. *J. ACM*, 67(5), sep 2020.

[CIS] CISCO. Snort intrution prevention system. `https://www.snort.org`.

[DA18] Javad Darivandpour and Mikhail J Atallah. Efficient and secure pattern matching with wildcards using lightweight cryptography. *Computers & Security*, 77:666–674, 2018.

[DdSGOT21] Cyprien Delpech de Saint Guilhem, Emmanuela Orsini, and Titouan Tanguy. Limbo: Efficient zero-knowledge mpcith-based arguments. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3022–3036, 2021.

[DSZ] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium*.

[FKL$^+$21] Nicholas Franzese, Jonathan Katz, Steve Lu, Rafail Ostrovsky, Xiao Wang, and Chenkai Weng. Constant-overhead zero-knowledge for ram programs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 178–191, New York, NY, USA, 2021. Association for Computing Machinery.

[Fri09] Keith B Frikken. Practical private dna string searching and matching through efficient oblivious automata evaluation. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 81–94. Springer, 2009.

[GAZ$^+$21] Paul Grubbs, Arasu Arun, Ye Zhang, Joseph Bonneau, and Michael Walfish. Zero-knowledge middleboxes. Cryptology ePrint Archive, Paper 2021/1022, 2021. `https://eprint.iacr.org/2021/1022`.

[GKWY20] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy*, pages 825–841. IEEE, 2020.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[GMW19] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pages 307–328. 2019.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[GSZ20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II*, pages 618–646. Springer, 2020.

[HL08] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference*, pages 155–175. Springer, 2008.

[HM18] Paul Hoffman and Patrick McManus. Dns queries over https (doh). Technical report, 2018. `https://www.rfc-editor.org/rfc/rfc8484`.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Crypto*, volume 2729, pages 145–161. Springer, 2003.

[Ker06] Florian Kerschbaum. Practical private regular expression matching. In *Security and Privacy in Dynamic Environments: Proceedings of the IFIP TC-11 21st International Information Security Conference*, pages 461–470, 2006.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved ot extension for transferring short secrets. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference*, pages 54–70, 2013.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 818–829, 2016.

[KM14] Franziskus Kiefer and Mark Manulis. Zero-knowledge password policy checks and verifier-based pake. In *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security*, pages 295–312. Springer, 2014.

[KRT18] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. Swim: Secure wildcard pattern matching from ot extension. In *International Conference on Financial Cryptography and Data Security*, pages 222–240. Springer, 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008*, pages 486–498. Springer, 2008.

[Kuk92] Karen Kukich. Techniques for automatically correcting words in text. *Acm Computing Surveys (CSUR)*, 24(4):377–439, 1992.

[LJA^{+}22] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. ppsat: Towards two-party private sat solving. In *31st USENIX Security Symposium*, pages 2983–3000, 2022.

[LW13] Peeter Laud and Jan Willemson. Universally composable privacy preserving finite automata execution with low online and offline complexity. 2013. `https://eprint.iacr.org/2013/678`.

[McC04] Martha M McCarthy. Filtering the internet: The children's internet protection act. *Educational Horizons*, 82(2):108–113, 2004.

[MNSS12] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *Cryptographers' Track at the RSA Conference*, pages 398–415. Springer, 2012.

[Moc87] Paul V Mockapetris. Domain names-concepts and facilities. Technical report, 1987. `https://www.rfc-editor.org/rfc/rfc1034`.

[NP99] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 245–254, 1999.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 129–139, 1999.

[pi-20] Regex Filters for Pi-hole. `https://github.com/mmotti/pihole-regex/blob/master/regex.list`, 2020.

[pi-22] Pi-hole:Network-wide protection. `https://pi-hole.net/`, 2022.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. 2005. `https://eprint.iacr.org/2005/187`.

[SBB19] Mohammad Hasan Samadani, Mehdi Berenjkoob, and Marina Blanton. Secure pattern matching based on bit parallelism. *International Journal of Information Security*, 18(3):371–391, 2019.

[SHd+14] Hirohito Sasakawa, Hiroki Harada, David duVerle, Hiroki Arimura, Koji Tsuda, and Jun Sakuma. Oblivious evaluation of non-deterministic finite automata with application to privacy-preserving virus genome detection. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 21–30, 2014.

[SLPR15] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM conference on special interest group on data communication*, pages 213–226, 2015.

[SP03] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 262–271, 2003.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[SW+21] Sheng Sun, Dr Wen, et al. zk-fabric, a polylithic syntax zero knowledge joint proof system. *arXiv preprint arXiv:2110.07449*, 2021.

[Tho68] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[Too] Aivo Toots. Zero-knowledge proofs for business processes. `https://cyber.ee/uploads/aivo_toots_msc_438a500a90.pdf`.

[TPKC07] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient dna searching through oblivious automata. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 519–528, 2007.

[Ver11] Damien Vergnaud. Efficient and secure generalized pattern matching via fast fourier transform. In *International Conference on Cryptology in Africa*, pages 41–58. Springer, 2011.

[VL06] Jan Van Lunteren. High-performance pattern-matching for intrusion detection. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–13. Citeseer, 2006.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 215–226, 2014.

[XYA+08] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, Geoff Hulten, and Ivan Osipkov. Spamming botnets: signatures and characteristics. *ACM SIGCOMM Computer Communication Review*, 38(4):171–182, 2008.

[Yao82] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[YMMP16] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M Pujol. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web*, pages 121–132, 2016.

[YSK$^+$13] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, pages 65–76, 2013.

[YWL$^+$20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated ot with small communication. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1626, 2020.

[ZE13] Samee Zahur and David Evans. Circuit structures for improving efficiency of security and privacy tools. In *2013 IEEE Symposium on Security and Privacy*, pages 493–507, 2013.

[ZMM$^+$20] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1919–1938, 2020.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. In *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

A TNFA Construction and Proofs

In this section, we provide a detailed description of the Thompson NFA algorithm and prove its related lemmas.

A.1 Thompson NFA

We describe the details of Thompson NFA construction in Algorithm 8.

A.2 Proofs of Lemmas

Lemma A.1 *For any two cross edges in E_ε, they are both forward edges. In other words, no edge "crosses" any backward edge in the TNFA graph.*

Proof: No backward edge crosses any other edge in the TNFA graph by construction (Algorithm 8). We can show this property hold for all regular expressions by giving an induction proof using the recursive regex definition. Given two regular expressions re, re satisfying this property, we have:

1. concatenation: the regular expression $(\mathsf{re}\ \mathsf{re}')$ just introduces a "new" forward edge in the TNFA graph, from the end node of re TNFA to the start node of re' TNFA. Hence the TNFA for $(\mathsf{re}\ \mathsf{re}')$ satisfies this property too.
2. union: The TNFA for $(\mathsf{re}|\mathsf{re}')$ introduces new forward edges, but they all have end points being either the first node or the last node of re or re' TNFA. Hence no backward edge in re' or re' crosses any of the newly introduced forward edges.
3. loop: $(\mathsf{re})^*$, introduces a new forward edges and a single backward edges - all of them have end points being either the first node or the last node of re or re' TNFA.

Proof of Lemma 3.1
Proof: It is sufficient to show that if there exists a path from u to v containing $t > 1$ backward edges, we can always find an alternative path from u to v that contains only $t - 1$ backward edges. Let the path be

Algorithm 8: Thompson NFA construction

```
Input: $\mathsf{re} \in \Sigma^n$
Output: $G(V,E), V = 0,\cdots,n, E, \mathsf{re} \in \Sigma^n$
1  Initialize an empty stack $'\mathsf{ops}$ of integers
2  Initialize a graph $G(V, E = E_\Sigma \cup E_\varepsilon)$ with nodes $\{0,\cdots,n\}$, Both $E_\Sigma$ and $E_\varepsilon$ are initialized to be
   empty
3  for $i = 0$ to $n$ do
4      $\mathsf{lp} \leftarrow i$
5      if $\mathsf{re}_i \overset{?}{=} '(' \vee \mathsf{re}_i \overset{?}{=} '|'$ then
6          $\mathsf{ops.push}(i)$
7      else
8          if $\mathsf{re}_i \overset{?}{=} ')'$ then
9              $\mathsf{or} \leftarrow \mathsf{ops.pop}()$
10             if $\mathsf{re}_{\mathsf{or}} \overset{?}{=} '|'$ then
11                 $\mathsf{lp} \leftarrow \mathsf{ops.pop}()$
12                 add $(\mathsf{lp}, \mathsf{or}+1)$ and $(\mathsf{or}, i)$ to $E_\varepsilon$
13             else
14                 if $\mathsf{re}_{\mathsf{or}} \overset{?}{=} '('$ then
15                     $\mathsf{lp} \leftarrow \mathsf{or}$
16     if $i < n-1 \wedge \mathsf{re}_{i+1} \overset{?}{=} '*'$ then
17         add $(\mathsf{lp}, i+1)$ and $(i+1, \mathsf{lp})$ to $E_\varepsilon$
18     if $\mathsf{re}_i \overset{?}{=} '(' \vee \mathsf{re}_i \overset{?}{=} '*' \vee \mathsf{re}_i \overset{?}{=} ')'$ then
19         add $(i, i+1)$ to $E_\varepsilon$
20 for $i = 0$ to $n-1$ do
21     if $\mathsf{re}_i \in \Sigma$ then
22         label the edge $(i, i+1)$ by $\mathsf{re}_i$ and add it to $E_\Sigma$
```

$P = (P'||u_0, \cdots u_h, w_0, \cdots, w_k, v_0, \cdots, v_\ell)$ where $\{u_i\}$, v_i, $\{w_i\}$ are all ascending lists, (u_h, w_0), (w_k, v_0) are backward edges, and $v_\ell = v$. We now discuss all three possible cases:

- $w_0 < u_h \le v_0 < w_k$: (u_h, w_0) and (w_k, v_0) are neither crossed nor nested. Then $(w_0 \le v_0 < w_k)$. Therefore, there exists $0 \le c \le k$ an edge $(w_c \le v_0 < w_{c+1})$. Furthermore, w_c should be equal to v_0, otherwise the edge (w_c, w_{c+1}) crosses the backward edge (w_k, v_0). Therefore, the path $P = (P'||u_0, \cdots u_h, w_0, \cdots, w_c = v_0, \cdots, v_\ell)$ is also a path from u to v with $t-1$ backward edges.

- $v_0 \le w_0 < u_h \le w_k$: (u_h, w_0) is nested within (w_k, v_0) are neither crossed nor nested. we thus have $w_0 < u_h \le w_k$. Then there exists $0 \le c \le k$ an edge $(w_c < u_h \le w_{c+1})$. Furthermore, w_{c+1} should be equal to u_h, otherwise the edge (w_c, w_{c+1}) crosses the backward edge (u_h, w_0). Therefore, the path $P = (P'||u_0, \cdots u_h = w_{c+1}, w_{c+2}, \cdots, w_k = v_0, \cdots, v_\ell)$ is also a path from u to v with $t-1$ backward edges.

- $w_0 \le v_0 < w_k \le u_h$: (w_k, v_0) is nested within (u_h, w_0). Using an argument similar to that before, we have $v_0 = w_c$ for some c. Thus, the path $P = (P'||u_0, \cdots u_h, w_0, \cdots, w_c = v_0, \cdots, v_\ell)$ is also a path from u to v with $t-1$ backward edges.

Proof of Theorem 3.1

Proof: The only difference between the transition TNFA simulation and 2 linear scan algorithm is in the epsilon transition step. To show correctness, we essentially need to prove, if a node v is reachable from an active node u at the start of the epsilon transition step, then s_v is set to 1 by the end of the epsilon transition step. From Lemma 3.1 there exists a path P with at max one backward edge from u to v in $G(V, E_\varepsilon)$.

1. Case 1: If P contains no backward edge:
   The path from u to v consists of only forward edges. Let the path be $u = u_1, u_2, \ldots, u_k = v$, where we have $_i < u_i$ if $i < j$. Hence in the iteration $t = 0$ on Step 13 of the algorithm, we know s_{u_2} is set to 1, since its predecessor s_{u_1} is set to 1. Inductively, s_{u_t} for any t is set to 1 since $s_{u_{t-1}}$ is set to 1.

2. Case 2: If P contains 1 backward edge:
   Let the one backward edge on the path P be (w, z).
   Case 2.1: $u \neq w$: There exists a path from u to w consisting only of forward edges, and hence s_w is set to 1 by the end of the first iteration of the For loop on Step 13. Variable s_z is set to 1 during the 2^{nd} iteration of the For loop, which will further help set s_v to 1 in the same iteration, since there is a path of forward edges from node z to v.
   Case 2.2: $u = w$: Since $z < u$, s_z is set to 1 during the first iteration of the For loop in Step 13, which would further set s_v to 1 in the same iteration since there exists a path consisting of forward edges from z to v.

Proof of Lemma 5.1
Proof: This follows from the construction of Thompson's NFA. Introduction of each | operator, let's say for regular expressions $(\mathsf{re}|\mathsf{re}')$, adds two long forward edges in the graph - one from the new opening parenthesis to the node labeled | and the other one from last node in re to the closing parenthesis of this regular expression. Hence if the last node of re has no other long forward outgoing edge (which can be proved by induction), then the Lemma holds for each new introduction of the union operator. Similarly, the introduction of each closure operator $*$, let's say for regular expressions $(\mathsf{re}*)$, adds a long forward edge from the opening parenthesis to the node labeled $*$, and it adds a long backward edge from the closing parenthesis to the opening parenthesis. Hence the lemma holds in this case too.

Proof of Lemma 5.2
Proof: The lemma can be obtained directly as a corollary of Lemma A.1.

Proof of Lemma 5.3
Proof: We can prove this by induction, similar to the proof of Lemma A.1. We can show this property hold for all regular expressions by giving an induction proof using the recursive regex definition. Given two regular expressions re, re satisfying this property, we have:

1. concatenation: the regular expression $(\mathsf{re}\ \mathsf{re}')$ just introduces a "new" forward edge in the TNFA graph, from the end node of re TNFA to the start node of re$'$ TNFA. Hence the TNFA for $(\mathsf{re}\ \mathsf{re}')$ satisfies this property too.
2. union : The TNFA for $(\mathsf{re}|\mathsf{re}')$ introduces a new pair of forward cross edges. Where one edge is from the first node of re to the first node of re$'$, and the other edge is from the node labeled | to the last node of re$'$. Hence the induction property holds.
3. loop: $(\mathsf{re})^*$, introduces a new forward edges and a single backward edges - all of them have end points being either the first node or the last node of re or re$'$ TNFA. Hence no new cross edges are introduced.

Algorithm 9: Zero-knowledge version of the Algorithm 2 (ZK-Regex)

```
   Private Input: $[\mathbf{x}] = ([\mathbf{x}_0] \dots [\mathbf{x}_{m-1}]), \mathbf{x} \in \Sigma^m$
   Public Input: TNFA $N = (G(V, E_\varepsilon), \mathsf{re})$
   Output: String holder and pattern holder shares $N(\mathbf{x})$
1  Function 2ScanTNFAEval($[\mathbf{x}], N$):
2      $[\mathbf{s}] \leftarrow ([1], [0], \dots, [0])$ where $\mathbf{s} \in \{0,1\}^{n+1}$
3      EpsilonTransitions($G(V, E_\varepsilon), [\mathbf{s}]$)
4      for $i = 0$ to $(m-1)$ do
5          CharacterTransitions($[\mathbf{x}_i], \mathsf{re}, [\mathbf{s}]$)
6          EpsilonTransitions($G(V, E_\varepsilon), [\mathbf{s}]$)
7      return $[s_n]$
8  Function CharacterTransitions($[\mathbf{x}_i], \mathsf{re}, [\mathbf{s}]$):
9      for $j = n$ to $1$ do
10         $[s_j] \leftarrow ([s_{j-1}] \wedge (\mathsf{re}_j \overset{?}{=} [\mathbf{x}_i])) \vee (\mathsf{re}_j \overset{?}{=} '?')$
11     $[s_0] \leftarrow 0$
12 Function EpsilonTransitions($G(V, E_\varepsilon), [\mathbf{s}]$):
13     for $t = 0$ to $1$ do
14         for $j = 0$ to $n$ do
15             $[s_j] \leftarrow [s_j] \vee (\vee_{(\alpha,j) \in E_\varepsilon} [s_\alpha])$
```

B ZK-Regex Protocol Description

In Section 4.1, we provide a brief introduction of the circuit

$$\mathcal{C}_{\mathsf{tnfa}}(\mathbf{x}, G(V, E_\epsilon), \mathsf{re}) \rightarrow s_n$$

that is translated from the TNFA Algorithm 2. It represents a statement that a private string $\mathbf{x}$ can be matched by a public regex $(G(V, E_\epsilon), \mathsf{re})$. The detailed ZK-Regex algorithm is shown in Algorithm 9. We use the notation $[\cdot]$ to represent a private witness in ZKP.

The circuit takes a private input $[\mathbf{x}]$ and a public input $(G(V, E_\epsilon), \mathsf{re})$. It sets the initial states $[\mathbf{s}]$ and starts to activate the states using the character transitions and epsilon transitions. Finally it outputs the final state $[s_n]$ for $s_n \in \{0, 1\}$ indicating unmatch or match.

C Two-Party Epsilon Transition: Instantiation and Security Proofs

In this section, we first provide the security proof of 1-out-of-N OT. Then we explain the detailed instantiation, security proof and optimizations of 1-out-of-N ROT. They are the building blocks for our OT-based epsilon transition protocols. At last, we provide the security proof of the OT-based and stack-based epsilon transition shown in Section 5.

C.1 Security Proof of 1-out-of-N OT

The Algorithm 7 realizes the 1-out-of-N oblivious transfer and it only depends on 1-out-of-N random oblivious transfer. It shares the same construction as the previous 1-out-of-N OT schemes [NP99; KKRT16; KK13]. We have the following theorem.

Theorem C.1 *The protocol shown in Algorithm 7 securely realizes the 1-out-of-N oblivious transfer in $\mathcal{F}_{\mathsf{OT}}$ with semi-honest security in the $\mathcal{F}_{\mathsf{OT}}$-hybrid mode.*

Algorithm 10: 1-out-of-N ROT protocol

```
Input: Define parameter $d := \lceil \log N \rceil$, security parameter $\kappa$ and message length $\ell$. Assume a
       correlation robust hash function $H$ and a function $G : \{0,1\}^\kappa \to \{0,1\}^\ell$.
Output: $P_0$ outputs $(m_0, \ldots, m_{N-1}) \in \{0,1\}^{N\times\ell}$. $P_1$ outputs $(b, m_b)$ where $b \in [0, N)$
1 For $i \in [d]$, $P_0$ and $P_1$ send $(\mathsf{rot}, 2, \kappa)$ to $\mathcal{F}_{\mathsf{OT}}$, which returns $(y_i^0, y_i^1)$ to $P_0$ and $(x_i, y_i^{x_i})$ to $P_1$.
2 $P_0$ and $P_1$ sets $a_0^0 = \mathbf{0}^\kappa$.
3 for $i = 1$ to $d$ do
4     for $j = 0$ to $2^{i-1} - 1$ do
5         $P_0$ computes $(a_i^{2j}, a_i^{2j+1}) := (H(a_{i-1}^j, y_i^0), H(a_{i-1}^j, y_i^1))$.
6     $P_1$ computes $k_i = \sum_{t=1}^{i} x_t \cdot 2^{i-t}$ and $a_i^{k_i} := H(a_{i-1}^{\lfloor k_i/2 \rfloor}, y_i^{x_i})$.
7 $P_0$ outputs $\{G(a_d^i)\}_{i\in[0,N)}$ and $P_1$ outputs $(k_d, G(a_d^{k_d}))$.
```

Proof: Let $\mathcal{A}$ be a PPT adversary who corrupts either P_0 or P_1. We construct a simulator $\mathcal{S}$ who simulates the view of the adversary. In the following, we consider two separate cases in which $\mathcal{A}$ corrupts P_0 then P_1.

Assume a $\mathcal{A}$ that corrupted P_0. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$. Upon receiving the rot command, it samples uniform $r_0, \ldots, r_{N-1} \in \{0,1\}^\ell$ and sends them to $\mathcal{A}$. It also sends a random index $\delta \in [0, N)$ to $\mathcal{A}$. During the online phase, $\mathcal{S}$ receives $\mathbf{y}$ from $\mathcal{A}$. Assume a $\mathcal{A}$ that corrupted P_1. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$. Upon receiving the rot command, it samples uniform $r \in \{0,1\}^\ell$ and $\gamma \in [0, N)$, and sends them to $\mathcal{A}$. Upon receiving δ, it computes $\alpha = \delta + \gamma$. During the online phase, $\mathcal{S}$ samples random $\mathbf{y} \in \{0,1\}^{N\times\ell}$ and sends it to $\mathcal{A}$.

In both cases, the views of $\mathcal{A}$ are perfectly simulated, assuming that the underlying 1-out-of-ROT is secure against semi-honest adversary.

C.2 Instantiating 1-out-of-N ROT

The details of the 1-out-of-N ROT protocol is shown in Algorithm 10. It depends on 1-out-of-2 ROT and a correlation robust hash (CRH) function [IKNP03]. The 1-out-of-2 ROT is instantiated by the vector oblivious linear evaluation (VOLE) based on pseudorandom correlation generator (PCG), which incurs communication < 1 bit per ROT in average [BCG+19a; YWL+20; BCG+19b]. At a high level, it first executes 1-out-of-2 ROT, which returns $d = \lceil \log_2 N \rceil$ 1-out-of-2 ROT messages. Based on these messages, P_0 builds a depth-d binary tree in which each path is associated with a combination of ROT messages. P_1 can only recover the nodes in one path depending on its choice index. Define the leaf nodes to be the 1-out-of-N ROT messages, P_0 outputs all leaf nodes and P_1 outputs only the b-th node. We define the following theorem and provide its proof.

Theorem C.2 *The 1-out-of-N random oblivious transfer protocol (in Figure 10) securely realizes the ROT function from the functionality $\mathcal{F}_{\mathsf{OT}}$ with semi-honest security in the $\mathcal{F}_{\mathsf{ROT}}$-hybrid model.*

Proof: The protocol starts with an invocation of $\mathcal{F}_{\mathsf{OT}}$ followed by only one-round communication from P_0 to P_1. We briefly sketch the proof by simulating the views for corrupted P_0 and P_1. First define a simulator $\mathcal{S}$ and an adversary $\mathcal{A}$ who controls P_0. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$. It samples uniform $\{(y_i^0, y_i^1)\}_{i\in[0,d)}$ and sends them to $\mathcal{A}$. The simulated view of $\mathcal{A}$ is indistinguishable from its view in the real world. Also, define a simulator $\mathcal{S}$ and an adversary $\mathcal{A}$ who controls P_1. $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$. It samples uniform $\{(x_i, y_i^{x_i})\}_{i\in[0,d)}$ and sends them to $\mathcal{A}$. A adversary $\mathcal{A}$ would succeed it if makes query to CRH with the same input that P_0 makes to compute nodes other than $a_d^{k_d}$. Define the number of queries that a real-world adversary $\mathcal{A}$ makes to CRH to be Q, the view of $\mathcal{A}$ is indistinguishable from the view of honest P_1 in the real-world except with probability $Q/2^\kappa$.

Batch 1-out-of-n+1 ROTs in epsilon transition. It appears multiple times in Algorithm 6 that the pattern holder and string holder invoke multiple 1-out-of-n+1 OTs with the same OT receiver's choice. We take advantage of this to further optimize the batch generation of ROT, which reduces the number of invocation of 1-out-of-n ROT by a factor of m (length of the string). In detail, to batch t 1-out-of-n+1 ROT with the same receiver's choice, two parties instantiate the function G defined in Algorithm 10 with a pseudorandom generator $\mathsf{PRG} : \{0,1\}^\kappa \to \{0,1\}^{t\times\ell}$. Each output of PRG is equally split into t segments and for $i \in [t]$, the i-th segment of size ℓ bits belongs to the i-th 1-out-of-n+1 ROT.

C.3 Security Proof of Algorithm 6

Proof of Theorem 5.2. The protocol shown in Algorithm 6 securely realizes the function $\mathsf{EpsilonTransition}$ described in Algorithm 2 (Lines 12-15) against semi-honest adversaries in the $(\mathcal{F}_{\mathsf{2PC}}, \mathcal{F}_{\mathsf{OT}})$-hybrid mode.
Proof: We follow the simulation-based paradigm to prove the Theorem 5.2. We first consider a corrupted string holder $\mathcal{A}$, who acts as the sender in $\mathcal{F}_{\mathsf{OT}}$ and a garbler in $\mathcal{F}_{\mathsf{2PC}}$. A simulator $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{2PC}}$. When emulating OT, it receives and stores the OT input messages from $\mathcal{A}$. When simulating the 2PC functionality, it receives the inputs from $\mathcal{A}$, samples a random bit $b \in \{0,1\}$ and returns it to $\mathcal{A}$. We consider a corrupted pattern holder $\mathcal{A}$. A simulator $\mathcal{S}$ emulates the functionality $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{2PC}}$. When emulating OT, it receives and stores the choice indices from $\mathcal{A}$ and returns a random bit c as the output for OT receiver. When simulating the 2PC functionality, it receives the inputs from $\mathcal{A}$, samples a random bit $d \in \{0,1\}$ and returns it to $\mathcal{A}$. Both views of the corrupted string holder and corrupted pattern holder are perfectly simulated.

C.4 Security Proof of Algorithm 5

Proof of Theorem 5.1. The protocol shown in Algorithm 5 securely realizes the function $\mathsf{EpsilonTransition}$ described in Algorithm 2 (Lines 12-15) against semi-honest adversaries in the $\mathcal{F}_{\mathsf{2PC}}$-hybrid mode.
Proof: In Algorithm 5, all components are executed in a garbled circuits protocol so the security fully relies on the underlying half-gates construction instantiating $\mathcal{F}_{\mathsf{2PC}}$. Define the computation of the algorithm 5 to be a binary circuit $\mathcal{C}$. Define a PPT adversary $\mathcal{A}$ and construct a simulator $\mathcal{S}$ that simulates views for $\mathcal{A}$ when $\mathcal{A}$ corrupts one of the parties. In both cases, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{2PC}}$ to receive $\mathcal{C}$ along with the input of $\mathcal{A}$, then sample a random output of the same length as $\mathcal{A}$'s output for $\mathcal{A}$.

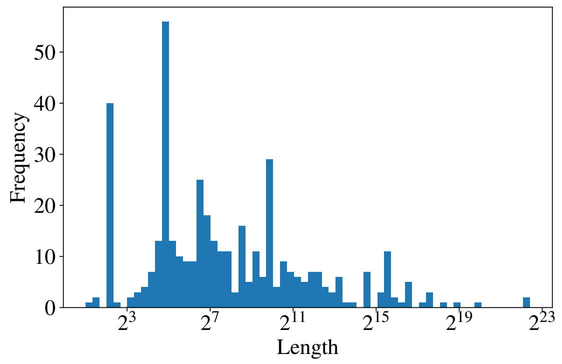

Figure 11: The distribution of lengths of regular expressions in SNORT. 85.5% expressions have length less or equal than 2^{12} bytes.

D Regular Expressions from SNORT

SNORT PCRE is a sophisticated form of regular expressions with advanced features like backreferences and recursive patterns. Our focus is on classical regular expressions, which typically support only disjunction, concatenation, and loop operations. We rewrite the SNORT PCRE into our format described in Section 2.1. This results in 416 SNORT PCRE that are suitable for use in our secure-regex protocol, with 356 (85.5%) of them having a length of $\leq 2^{12}$ bytes. The details of their distribution are shown in Figure 11.