Accumulating Automata and Cascaded Equations Automata[⋆]
for Communicationless Information Theoretically Secure Multi-Party Computation
(Preliminary Report)

Shlomi Dolev[1] and Niv Giboa[2] and Ximing Li[1]

[1] Department of Computer Science, Ben Gurion University of Negev
dolev@cs.bgu.ac.il, ximing@post.bgu.ac.il
[2] Department of Communication Systems Engineering, Ben Gurion University of Negev
gilboan@cse.bgu.ac.il

Abstract. Information theoretically secure multi-party computation implies severe communication overhead among the computing participants, as there is a need to reduce the polynomial degree after each multiplication. In particular, when the input is (practically) unbounded, the number of multiplications and therefore the communication bandwidth among the participants may be practically unbounded. In some scenarios the communication among the participants should better be avoided altogether, avoiding linkage among the secret share holders. For example, when processes in clouds operate over streaming secret shares without communicating with each other, they can actually hide their linkage and activity in the crowd. An adversary that is able to compromise processes in the cloud may need to capture and analyze a very large number of possible shares.

Consider a dealer that wants to repeatedly compute functions on a long file with the assistance of m servers. The dealer does not wish to leak either the input file or the result of the computation to any of the servers. We investigate this setting given two constraints. The dealer is allowed to share each symbol of the input file among the servers and is allowed to halt the computation at any point. However, the dealer is otherwise stateless. Furthermore, each server is not allowed any communication beyond the shares of the inputs that it receives and the information it provides to the dealer during reconstruction.

We present a protocol in this setting for generalized string matching, including wildcards. We also present solutions for identifying other regular languages, as well as particular context free and context sensitive languages. The results can be described by a newly defined *accumulating automata* and *cascaded equations automata* which may be of an independent interest. As an application of *accumulating automata* and *cascaded equations automata*, secure and private repeated computations on a secret shared file among communicationless clouds are presented.

Keywords: Automata, Theoretically secure, Multi-Party Computation, Communicationless clouds

[⋆] Partially supported by Deutsche Telekom, Rita Altura Trust Chair in Computer Sciences, Lynne and William Frankel Center for Computer Sciences, Israel Science Foundation (grant number 428/11), Cabarnit Cyber Security MAGNET Consortium, Grant from the Institute for Future Defense Technologies Research named for the Medvedi of the Technion, and Israeli Internet Association.

1 Introduction

Secure multi-party computation (MPC) is a powerful concept in secure distributed computing. The goal of secure MPC is to enable a set of m mutually distrusting parties to jointly and securely compute a function f of their private inputs, even in the presence of a computationally unbounded active adversary Adv. For example, two millionaires can compute which one is richer, without revealing their actual worth. In secure MPC, two or more parties want to conduct a computation based on their private inputs, but neither party is willing to disclose its own input to anybody else.

Secure multi-party computation participants can compute any function on any input, in a distributed network where each participant holds one of the inputs, ensuring independence of the inputs, correctness of the computation, and that no information is revealed to a participant in the computation beyond the information that can be inferred from that participants' input and output [22, 14]. Like other cryptographic protocols, the security of MPC protocol can rely on different assumptions:

- It can be computational, namely, based on the common belief on the complexity of mathematical tasks such as factoring, or information theoretically secure which is unconditional secure, and usually based on secret sharing schemes [28, 5].
- The settings in which the scheme is described may differ, possibly assuming that participants use a synchronized network, that a secure and reliable broadcast channel exists, that a secure communication channel exists between every pair of participants, such that an adversary cannot tap-in, modify or generate messages in the channel, and alike.

Secure multi-party computation can be realized in various settings for computing general functions [21]. However, the general scheme presented in [21] may be impractical due to efficiency reasons, partly due to the communication required among the participants. In [2], Ben-Or et al demonstrated that any n-party functionality can be computed with perfect security in the private channels model. For the main result of [2], Gilad Asharov and Yehuda Lindell gave a full proof in [1] in 2011. Recently, several fast MPC protocols [23, 29, 24] are proposed to meet a specified security level at a relative low cost, while they all based the security of their protocols on computational unproven hardness assumptions. According to the experiment in [29], they can finish computing the output of one logical gate in about 4ms. Thus, simple searches will cost several minutes. Thereby, currently, it is impossible to use this kind of computing protocol to design practical software.

In this paper, we are concerned with communicationless information theoretically secure multi-party computation over long input streams. A dealer $\mathcal{D}$ may secretly share an initial value among the m servers (participants). Subsequently, the dealer is responsible for handling the input stream stream (or an input file) and distributing appropriate shares to the participants. We assume a stateless dealer, allowing the dealer to temporarily store the current input to the system, process the input and send (not necessarily simultaneously) secret shares of the inputs to the participants. Note, that one of the participants may act as the dealer, or the participants may alternate among themselves in serving as the dealer. In such a case one participant communicates with the rest to convey the input (shares), still the inherent quadratic complexity needed to reduce the polynomial degree in the classical information theoretically secure multi-party computation is avoided in our schemes. Moreover, in case the input symbols have been shared and assigned to the participants in the initialization phase, every participant can independently (and asynchronously) process the shares of the input, and sends the result when the global output has to be determined. For example assigning shares of a file up-front to participants to allow repeated search of patterns, without revealing neither the file nor the search result to the participants. No participant returns any information back during the execution of the algorithm. At any point in the execution the dealer may ask some participants to send their results back, then the dealer can reconstruct the actual result of the algorithm.

Related work We firstly describe the related results concerning multi-party computation. Then, we turn to works that suggest outsourcing finite automata, perennial distributed computation on common inputs and secure computation on stream of data. At last we address and compare our results with fully homomorphic encryption schemes.

Multi-party computation Josh Cohen Benaloh describes the homomorphism property of Shamir's linear secret sharing scheme [3], with the help of communication to decrease the polynomial degree. Ronald Cramer et al. presented a method [7] for converting shares of a secret into shares of the same secret in a different secret-sharing scheme using only local computation and no communication between players. They showed how this can be combined with any pseudorandom function to create, from initially distributed randomness, any number of Shamir's secret-shares of (pseudo)random values without communication. Damgard et al. showed how to effectively convert a secret-shared bit over a prime field to another field [8]. By using a pseudorandom function, they showed how to convert arbitrary many bit values from one initial random replicated share.

Outsourcing finite automata In [31], Brent Waters provides a functional encryption system that supports functionality for regular languages. In this system a secret key is associated with a deterministic finite automaton (DFA) M. A ciphertext, ct, encrypts a message msg associated with an arbitrary length string w. A user is able to decrypt the ciphertext ct if and only if the automaton M associated with his private key accepts the string w. Motivated by the need to outsource file storage to untrusted clouds while still permitting limited usage of that data by third parties, the work in [25] presented practical protocols by which a client (the third-party) can evaluate a DFA on an encrypted file stored at a server (the cloud), once authorized to do so by the file owner. All the above schemes are based on unproven, commonly believed to be hard mathematical tasks and are not information theoretically secure.

Perennial distributed computation on common inputs In 2007, Dolev et al. [12, 13] presented the settings for infinite private computation and presented few functions that can operate under a global input. Then in 2009, Dolev et al. [9] presented schemes that support infinite private computation among participants, implementing an oblivious universal Turing machine. At each single input of the machine, participants need to broadcast information in order to reduce the degree of the polynomial used to share secrets. Based on combination of secret-sharing techniques and the Krohn-Rhodes decomposition of finite state automata, Dolev et al. [10] proposed the first communicationless scheme for private and perennial distributed computation on common inputs in a privacy preserving manner, assuming that even if the entire memory contents of a subset of the participants are exposed, no information about the state of the computation is revealed. The scheme in [10] does not assume a priori bound on the number of inputs. However, the scheme assumes a global input which reveals information on the computation and the computational complexity of the algorithm of each participant is exponential in the automata number of states. Relying on the existence of one-way functions or common long one time pads, Dolev et al. [11] showed how to process a priori unbounded number of inputs for inputs over a finite state automaton (FSA) at a cost that is linear in the number of FSA states. Although the authors can hide the current state of the FSA, the dealer must supply the input symbols in plain text to each participant.

Secure computation on data stream Private stream searching. In [26], Ostrovesky et al. defined the problem of private filtering where a data stream is searched for predefined keywords. The schemes are also implemented by Paillier homomorphic cryptosystem. The proposed scheme was improved by John Bethencourt et al. in [4] reducing the communication and storage complexity. Our scheme suggests framework for richer tasks and is information theoretically secure rather than computationally secure.

Fully homomorphic encryption In his seminal paper [16], Craig Gentry presented the first fully homomorphic encryption (FHE) scheme which is capable of performing encrypted computation on Boolean circuits. A user specifies encrypted inputs to the program, and the server computes on the encrypted inputs without gaining information concerning the input or the computation state. Following the outline of Gentry's, many subsequent FHE schemes [17, 30, 6, 19] are proposed and some of which are even implemented [18]. Most recently, Craig Gentry et al. executed one AES-128 encryption homomorphically in eight days [20]. Most encrypted computation examples are restricted to fixed-iteration loops or Boolean circuits, however, Fletcher et al. try to build a compiler for encrypted computation of general programs [15]. Fletcher et al. also formally show how a Turing machine operation can be transformed into an arithmetic circuit that can be evaluated under encryption. The FHE schemes that follow the outline of Gentry's original construction are inefficient in that their per-gate computation over-head is a large polynomial in the security parameter and are furthermore only computationally secure.

Our contribution We assume there is one dealer $\mathcal{D}$ who wants to perform secure private computation over a very long input stream which may be actually unbounded. The dealer uses m cloud servers or agents $\mathcal{P}_1, \ldots, \mathcal{P}_m$ which perform a computation over the input stream received from $\mathcal{D}$. The dealer $\mathcal{D}$ sends different input shares to every agent. Agents do not communicate with each other. Any agent cannot learn anything about the original inputs that $\mathcal{D}$ partitions to shares, as the dealer uses Shamir's secret sharing to partition any symbol of the original input to be sent to the agents. At any given stage the dealer $\mathcal{D}$ may collect the state of the agents and obtain the computation result. The agents use memory that is logarithmic in the length of the input, and therefore can accommodate practically unbounded inputs. We present two types of solutions, by introducing two new automata, the first one is *accumulating automata* and the second one is *cascaded equations automata*.The two types of automata are practical tools for communicationless MPC and can be implemented on equipments with low compuation capability, as all the compuations consist of additions and multiplications over small finite field.

Outline of the paper In Section 2, we exploit the method to implement communicationless secure and private multi-party computation for any string matching. This string matching example is generalized in Section 3 to the definition of accumulating automata and DAG accumulating automata which can be used to implement algorithms securely and privately. Then, in Section 4, we discuss the relationship between DAG accumulating automata, secret sharing and secure multi-party computations. Any algorithm that can be implemented by a DAG accumulating automaton can be secret shared, and then be securely and privately executed among constant number of participants. Other applications of accumulating automata beyond string matching are proposed in Section 5. For strictly unbounded input, in Section 6 we define a new kind of automata named cascaded equations automata and its application in the scope of secure multi-party computation over bounded *and* (strictly) unbounded inputs. We note however that the scope of languages recognizable by the cascaded equations automata is more restricted. An implementation of the accumulating automaton to obtain a secure computation over secret shared file among communicationless clouds is presented in Section 7. Conclusions and discussions appear in Section 8.

2 Secure private multi-party computation for string matching

String matching is a basic task used in a variety of scopes. A pattern (string) has to be found as part of text processing, also as part of malware (virus) defense, pattern recognition, bioinformatics and database query. The inputs are text and a pattern, the pattern is usually much shorter than the text. The goal is to find whether the pattern appears in the text or not. Fig. 1 describes a simple example of string matching. One brute force method for string matching is to check every single character as being the first character of the pattern and match the entire pattern against the text, starting at this character position. In this section, we introduce a new method to implement secure multi-party computation for string matching, where the text (and possibly the pattern) is unknown to the processing participants. Using the newly defined string algorithm over directed graph that is presented in Section 2.1, we propose a communicationless secure private multi-party string matching protocol in Section 2.2. More communicationless secure private multi-party string matching algorithms are given in Appendix 2.2.

PATTERN $LOVE$
↓
TEXT $ALICE \quad LOVES \quad BOB$

Fig. 1: String matching example

2.1 String matching algorithm over directed graph

We start describing a simplified non-secure version of the algorithm. Then we detail the way to obtain information theoretically secure computation extending the simplified version. Fig. 2 depicts a directed graph $\mathbb{G}$ which will be used to implement the string matching task of Fig. 1. $\mathbb{G}$ is used to check whether there is a substring $LOVE$ in the text. In $\mathbb{G}$, there are five nodes labeled N_1 to N_5 and four arcs labeled L, O, V, E, respectively. N_5 is a special node called *accumulating* node that is depicted by two concentric circles.

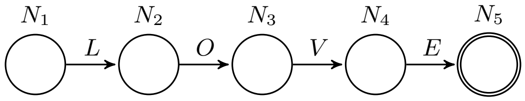


Fig. 2: String matching algorithm over directed graph $\mathbb{G}$

Initial marking of the graph At the initial stage, we assign an initial value for every node in the graph. N_1 is initially set to 1. N_2 to N_5 are initially set to 0. Namely,

$$N_1^{(0)} = 1, N_2^{(0)} = N_3^{(0)} = N_4^{(0)} = N_5^{(0)} = 0$$

Execution of the string matching algorithm Each input value assigns a new (integer) value to every node. $N_i^{(j)}$ denotes the value of the node N_i immediately after step j. According to the pattern, we define an input vector $\boldsymbol{v}$ in which each element matches one corresponding element in the pattern.

Since the pattern consists of four characters, $\{L, O, V, E\}$, we use a vector of four binary values that represents each possible character in the pattern. If the input character does not appear in the pattern, then the value of the vector $\boldsymbol{v}$ is set to $(0, 0, 0, 0)$. In particular, when the input symbol is O then the vector $\boldsymbol{v}$ is set to $(0, 1, 0, 0)$ and when the input symbol is, say C, then the vector $\boldsymbol{v}$ is set to $(0, 0, 0, 0)$. The value of N_1 is initialized to be 1 and is unchanged during the entire string matching process. For any given input vector (v_1, v_2, v_3, v_4), the values of *all* the marking of nodes of the graph are *simultaneously* computed as follows

$$\begin{array}{l} N_2^{(i+1)} = N_1^{(i)} \cdot v_1 \\ N_3^{(i+1)} = N_2^{(i)} \cdot v_2 \\ N_4^{(i+1)} = N_3^{(i)} \cdot v_3 \\ N_5^{(i+1)} = N_5^{(i)} + N_4^{(i)} \cdot v_4 \end{array} \tag{1}$$

Equation 1 defines the *transition* functions for the string matching algorithm. Note that N_5, being accumulated node, accumulates values, while the rest of the nodes recompute values based only on values of neighboring nodes.

Result of the algorithm At any time, we can check the value of the node N_5. If $N_5 > 0$, then we know, there is at least one match. Actually, the value of the node N_5 encodes the number of times the pattern has occurred in the input stream. Here we should assume that the number of occurrences does not exceed the maximal integer that the system can maintain and represent for N_5.

2.2 Communicationless secure private multi-party string matching protocol

Next we present a secure multi-party string matching algorithm using Shamir's secret sharing scheme to mimic the algorithm presented above. Among the whole protocol, the computation field is a big finite field. We assume all the computations will not overflow during the execution of the protocol.

Initial stage Nodes' values are shared among several participants using secret sharing and so are the entries of the vector that represent each symbol of the input text. For ease of discussion assume that the input symbols are represented by secret shares of polynomial of degree 1. Since the transition function includes multiplication, the degree of the polynomial that encodes the value of a certain node is one more than the degree of the preceding node. For our particular example, we have to use at least six participants to ensure that the result encoded in N_5 can be decoded.

For simplicity, we assume there are six participants $\mathcal{P}_1, \ldots, \mathcal{P}_6$ that undertake the task of multi-party computation string matching. For the five nodes of the graph, we define five random polynomials f_1 to f_5, where f_i is of degree i. We use each corresponding polynomial to secret share each node's initial value among the six participants, each partner $\mathcal{P}_i$ receives one share. We denote the initial share of the node N_j that is maintained by the participants $\mathcal{P}_i$ by $S^{(0)}_{\mathcal{P}_i,N_j}$.

Execution stage Each symbol α is mapped to an input vector $\boldsymbol{v}$. Then each element in the input vector $\boldsymbol{v}$ is secret shared into six parts by a random polynomial of degree 1. Each share of the input vector is then sent to one of the participants. For the participant $\mathcal{P}_i$, $1 \leq i \leq 6$, the corresponding shares of the input vector are denoted $(S_{i,v_1}, S_{i,v_2}, S_{i,v_3}, S_{i,v_4})$. We also secret share the number 1 into six shares by a random polynomial of degree 1. The six shares are denoted as follows

$$S_{1,v_0}, S_{2,v_0}, S_{3,v_0}, S_{4,v_0}, S_{5,v_0}, S_{6,v_0}$$

where the share S_{i,v_0} will be sent to the participant $\mathcal{P}_i$.

Immediately after processing the k^{th} input symbol, the value of the (share of) node N_j that is stored by the participant $\mathcal{P}_i$ is denoted $S^{(k)}_{\mathcal{P}_i,N_j}$. When the dealer sends a vector as follows

$$(S_{i,v_0}, S_{i,v1}, S_{i,v2}, S_{i,v3}, S_{i,v4})$$

$\mathcal{P}_i$ executes the following transitions:

$$\begin{aligned}
S^{(k+1)}_{\mathcal{P}_i,N_1} &= S_{i,v_0} \\
S^{(k+1)}_{\mathcal{P}_i,N_2} &= S^{(k)}_{\mathcal{P}_i,N_1} \cdot S_{i,v_1} \\
S^{(k+1)}_{\mathcal{P}_i,N_3} &= S^{(k)}_{\mathcal{P}_i,N_2} \cdot S_{i,v_2} \\
S^{(k+1)}_{\mathcal{P}_i,N_4} &= S^{(k)}_{\mathcal{P}_i,N_3} \cdot S_{i,v_3} \\
S^{(k+1)}_{\mathcal{P}_i,N_5} &= S^{(k)}_{\mathcal{P}_i,N_5} + S^{(k)}_{\mathcal{P}_i,N_4} \cdot S_{i,v_4}
\end{aligned}$$

Collection stage Whenever we want to compute the result of the algorithm, we ask all the participants to send the value that corresponds to N_5 back. Having the shares of all participants, we can construct the actual value of N_5 using Lagrange interpolation. The value obtained indicates whether the search is successful in finding the string or not.

Analysis of the system In the protocol, we have not discussed the computational field. Apparently, the greatest value is associated with the node N_5, this value represents the number of times the pattern was found in the text, namely, is bounded by the length of the input text. Thus, for every practical system a field that can be represented by a counter of, say, 128 bits will surely suffice.

Note that the participants do not know the inputs and the results during the entire execution of the string matching. One can also secure the pattern by executing such string matching over all possible strings, collect all results, and compute only the result of the pattern of interest.

2.3 Matching several strings simultaneously

The method discussed in the previous subsection is also good for simultaneous multiple strings matching, which means we can search more than one string simultaneously. An example of a directed graph for matching more than one string at the same time is described in Fig. 3.

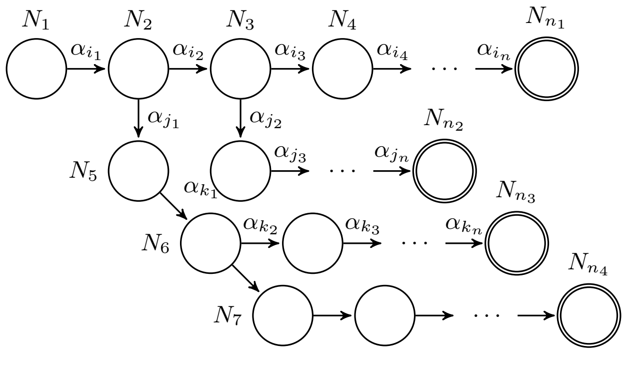


Fig. 3: A directed graph for multiple string matching

The *transition* function for this algorithm given in Equation 2 is similar to the one defined by Equation 1. In Equation 2, each node value is computed depending on the input and/or the previous state of the node. At step k, under each input, for each node N_i, we compute the next value as follows

$$N_i^{(k+1)} = \begin{cases} v_0 & \text{if } i = 0 \\ N_{i-1}^{(k)} \cdot v_i & \text{if } N_i \text{ is not an} \\ & \text{accumulating node} \\ N_i^{(k)} + N_{i-1}^{(k)} \cdot v_i & \text{if } N_i \text{ is an} \\ & \text{accumulating node} \end{cases} \tag{2}$$

2.4 General string matching

To allow any string matching, we need to address ways to implement the basic wildcard characters "?" and "*", as we detail next.

String matching algorithm with question mark in the pattern A character "?" is a character that may be substituted by any single character of all the possible characters. The directed graph for the matching algorithm that includes a question mark in the pattern is described in Fig. 4. The arc that represents the question mark is marked by the integer 1 and implies a transition that uses the marking of the previous node unchanged (multiplied by 1).

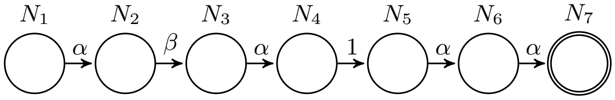


Fig. 4: String matching algorithm with one wildcard "?" in the pattern ($\alpha\beta\alpha?\alpha\alpha$)

The *transition* function for this algorithm given in Equation 3 is similar to the transition function defined by Equation 2. In step k, under each input, for each node N_i, we compute as follows

$$N_i^{(k+1)} = \begin{cases} v_0 & \text{if } i = 0 \\ N_{i-1}^{(k)} & \text{if the former edge is labeled by 1} \\ N_{i-1}^{(k)} \cdot v_i & \text{if } N_i \text{ is not an accumulating node;} \\ & \text{the former edge is not labeled by 1} \\ N_i^{(k)} + N_{i-1}^{(k)} \cdot v_i & \text{if } N_i \text{ is an accumulating node;} \\ & \text{the former edge is not labeled by 1} \end{cases} \tag{3}$$

String matching algorithm with a star wildcard in the pattern A wildcard character "*" is a character that may be substituted by any number of the characters from all the possible characters. The directed graph for the matching algorithm for a pattern with a star is described in Fig. 5.

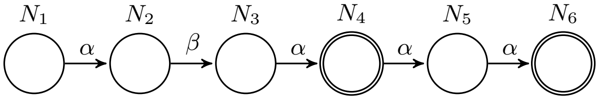


Fig. 5: String matching algorithm with one wildcard "*" in the pattern ($\alpha\beta\alpha * \alpha\alpha$)

The *transition* function for this algorithm given in Equation 4 is similar to the one defined by Equation 3. In step k, under each input, for each node N_i, we compute as follows

$$N_i^{(k+1)} =$$

$$\begin{cases} v_0 & \text{if } i = 0 \\ N_{i-1}^{(k)} \cdot v_i & \text{if } N_i \text{ is not an accumulating node} \\ N_i^{(k)} + N_{i-1}^{(k)} \cdot v_i & \text{if } N_i \text{ is an accumulating node} \end{cases} \quad (4)$$

2.5 Any secure private string matching algorithm

In subsection 2.1 we have shown how to perform a basic string matching algorithm on a directed graph. In subsection 2.2 we detailed a secure and private implementation of the algorithm in the scenario of multi-party computation without communication. Based on the basic implementation we present methods for implementing complicated string matching algorithms with wildcards in the pattern. Thus, we can implement (practically) any string matching algorithm securely and privately without communication between participants. The limitation of the value in the accumulating nodes is only theoretic, as for any text length n (say, even of practically not existing length of 2^{128} characters) and a pattern that yields l accumulating nodes, $l \cdot \log n$ bits are needed to encode a state. The field of the numbers should be n or (slightly) bigger.

3 Accumulating automata

In this section, we generalize the string matching scheme by defining general *accumulating automata (AA)* and then show ways to implement *dag accumulating automata* that are directed acyclic (not necessarily connected) graphs structure. Then, we define ways to *mark* the AAs and the corresponding semantics for such marking. At last we give a concrete example for DAA.

Accumulating automata are state-transition systems formally defined next:

Definition 1. *An accumulating automaton is a triple $\mathcal{A} = (V, \Sigma, T)$ where:*

- *V is a set of variables, called* nodes. *A node is either regular or accumulating. For an accumulating automaton, marking operation means to assign value for each node and updating process of an automaton means to refresh all the nodes on new input. On updating, the new marking of a regular node is a function of the marking of neighboring nodes and the inputs, while the marking computation function of an accumulating node also considers the node's previous value;*
- *Σ is the alphabet, which is the set of input symbols (or characters) that the automaton should operate on. 1 is defined as a special symbol to denote any symbol in the alphabet;*
- *T is a set of transitions. A triple $(p_1, \alpha, p_2) \in V \times \Sigma \to V$ is called a* transition *or* arc*, and is written $\delta(p_1, \alpha) = p_2$. For every δ in T there exist $p, q \in V$ and $\alpha \in \Sigma$ such that $\delta(p, \alpha) = q$.*

We remark that, one may consider a more expensive operation, where $\delta(p, r, \alpha) = q, p \cdot r \cdot \alpha = q$ (or even an operation that multiplies the marking of more than two nodes). This type of operation yields an addition of the *degree* of the polynomial used to secret share the node.

Accumulating automaton is represented by a (possibly disconnected) directed graph where each regular node is depicted by a circle, accumulating node by two concentric circles, and transitions by (directed) arcs. Input symbols are labeled as symbols above the corresponding transitions. Node q is called *free node*, if, for any node p and any input symbol α, no transition can fulfill $\delta(p, \alpha) = q$.

Definition 2. *(Dag Accumulating Automata (DAA)) Accumulating automaton that defines a graph $\mathbb{G}$ that is acyclic, namely, without cycles and self-loops is a* dag accumulating automaton. *In other words, dag accumulating automaton is an accumulating automaton for which it holds for any p in V, there does not exist $\alpha_1, \ldots, \alpha_n \in \Sigma$ and $\delta_1, \ldots, \delta_n \in T$, such that*

$$\delta_n(\cdots\delta_2(\delta_1(p, \alpha_1)), \alpha_2)\cdots) = p$$

Moreover, for every p and q in V, if there exists $\alpha \in \Sigma$ such that $\delta(p, \alpha) = q$ then $p \neq q$.

Definition 3. *(Marking of accumulating automata) A marking of an accumulating automaton $\mathcal{A} = (V, \Sigma, T)$ is a vector of values, one integer value for each node in V. A marked automaton $\mathcal{A}$ is a 4-tuple (V, Σ, T, M), where M is the marking vector.*

Definition 4. *(Execution semantics of AA) The behavior of an accumulating automaton is defined as a relation on its markings, as follows. Assume that immediately after the j^{th} input symbol, node p_i has the value $n_{p_i}^{(j)}$. On inputting the $(j+1)^{st}$ symbol α, for* all *the transitions $\delta(p_t, \alpha_i) = p_i$, where $p_t \in V$, $\alpha_i \in \Sigma$, the new value of p_i is computed as*

- *If p_i is a regular node, then*

$$n_{p_i}^{(j+1)} = \sum_{\substack{\delta(p_t, \alpha_i) = p_i, \\ \forall p_t \in V;\ \alpha = \alpha_i}} n_{p_t}^{(j)} + \sum_{\substack{\delta(p_t, \alpha_i) = p_i, \\ \forall p_t \in V;\ \alpha = 1}} n_{p_t}^{(j)}$$

- *If p_i is an accumulating node, then*

$$n_{p_i}^{(j+1)} = n_{p_i}^{(j)} + \sum_{\substack{\delta(p_t, \alpha_i) = p_i, \\ \forall p_t \in V;\ \alpha = \alpha_i}} n_{p_t}^{(j)} + \sum_{\substack{\delta(p_t, \alpha_i) = p_i, \\ \forall p_t \in V;\ \alpha = 1}} n_{p_t}^{(j)}$$

Marking and execution of accumulating automata We give a simple example of dag accumulating automaton $\mathcal{DAA}^{\alpha\beta\gamma}$=$(V, \Sigma, T)$ in Fig. 6. By checking the marking of this automaton, we can decide whether the input stream is $\alpha\beta\gamma$ or not. It means that we will use this dag accumulating automaton as deterministic finite automaton that checks whether the input language is $\alpha\beta\gamma$ or not. The four regular nodes are $V = \{N_1, N_2, N_3, N_4\}$. The input symbols are from the alphabet $\Sigma = \{\alpha, \beta, \gamma\}$. N_1 is a *free node* and is always assigned 0. The transitions are:

$$N_2 = \delta_1(N_1, \alpha);\ \ N_3 = \delta_2(N_2, \beta);\ \ N_4 = \delta_3(N_3, \gamma)$$

Initial marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ The initial marking of the automaton is:

$$N_1^{(0)} = 1;\ \ N_2^{(0)} = 0;\ \ N_3^{(0)} = 0;\ \ N_4^{(0)} = 0$$

The initial marking automaton is depicted in Fig 6.

Execution of the DAA $\mathcal{DAA}^{\alpha\beta\gamma}$ Executing the DAA means to retrieve symbols one by one from the input stream and input to the DAA. The input triggers transitions of the automaton resulting in a new marking. Assume the input symbol is α, then we set the input vector to $\boldsymbol{v} = (v_0, v_1, v_2, v_3) = (0, 1, 0, 0)$, and compute the new marking of the automaton.

The transitions are computed as follows

$$N_1^{(1)} = v_0 = 0;\ \ N_2^{(1)} = N_1^{(0)} \cdot v_1 = 1;\ \ N_3^{(1)} = N_2^{(0)} \cdot v_2 = 0;\ \ N_4^{(1)} = N_3^{(0)} \cdot v_3 = 0$$

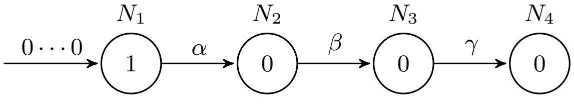


Fig. 6: Example of dag accumulating automaton $\mathcal{DAA}^{\alpha\beta\gamma}$ and the initial marking

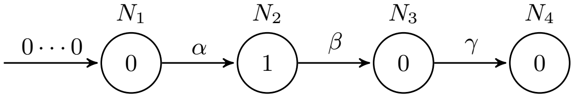


Fig. 7: Marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ after processing the input vector $\boldsymbol{v} = (0, 1, 0, 0)$ which represents the input α

Here, the new marking of the automaton is as in Fig 7.
Description of the marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ The marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ is changed by the input symbol that is sent by the dealer. At any time, we can check the marking of $\mathcal{DAA}^{\alpha\beta\gamma}$. If the marking is $(0, 0, 1, 0)$, we know the input stream is $\alpha\beta$. Accumulating automaton $\mathcal{DAA}^{\alpha\beta\gamma}$ can be used to accept the language $\alpha\beta\gamma$. The marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ reveals whether the input stream is accepted or not.
Correctness of $\mathcal{DAA}^{\alpha\beta\gamma}$ We will analyze the marking of the automaton under all possible input streams to check whether the automaton represents the function properly or not. Prior to the first input the marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ is $(1, 0, 0, 0)$ and the state of the automaton is "rejected". If (1) the first input symbol is not α; (2) the first input symbol is α, the second symbol is not β; (3) the first two input symbols are $\alpha\beta$, the third symbol is not γ, then the marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ is $(0, 0, 0, 0)$, and therefore the state of the automaton is "rejected". In all the three cases above, any successive additional input symbol will not change the marking of the automaton to $(0, 0, 0, 1)$, thus, implying that the whole input stream will be rejected.

In other words, if and only if the first three input symbols are $\alpha\beta\gamma$, then the marking of $\mathcal{DAA}^{\alpha\beta\gamma}$ is $(0, 0, 0, 1)$, and the state of the automaton is "accepted". Any extra input(s) will change the marking of the automaton to $(0, 0, 0, 0)$, and the state of the automaton is also changed to "rejected".

4 Dag accumulating automata and communicationless multi-party computation

Assume one dealer wants to execute DAA under a long input stream with the help of m servers without the leakage of the marking of the automaton and the whole input to the automaton. The dealer may secret share the marking of the DAA into m shares and assign each share to one of the servers. When the dealer wants to execute the DAA, she secretly shares each input into m shares and sends each share to a distinct server. Each server will manipulate its local DAA share and local input share to obtain a share of the new marking of the DAA. At some point, the dealer will ask all the servers to send shares back and use these shares to construct the current marking of the original DAA. An unprivileged subgroup of the servers will have no information concerning the inputs (but an upper bound on its length) and/or the computation result. The servers do not know, in terms of information theoretical security, the actual value of the input sequence and the marking of the DAA.

Before stating the relationship between DAA and secret sharing, we define a *route* and the *polynomial degree* of a *node* in (the graph $\mathbb{G}$ of) a DAA and define the *polynomial degree* of the entire DAA. We also define the *accumulating field* of a DAA. A sequence of nodes $\{N_{i_1}, \ldots, N_{i_{k+1}}\}$ is a *route*, if there are a sequence of transitions $\{\delta_{j_1}, \ldots, \delta_{j_k}\}$ and input symbols $\{\alpha_{t_1}, \ldots, \alpha_{t_k}\}$, such that

$$\delta_{j_1}(N_{i_1}, \alpha_{t_1}) = N_{i_2}, \cdots, \delta_{j_k}(N_{i_k}, \alpha_{t_k}) = N_{i_{k+1}}$$

The longest route always starts in a free node, i.e., a node with no incoming arcs. Let t be the secret sharing threshold, the minimal number of participants needed to reveal the automaton state, where $t-1$ is the polynomial degree in which the marking of the free nodes and the inputs are encoded.

Definition 5. *(*Polynomial degree *of a node and a DAA) Assuming t to be the secret sharing threshold, for any node N_i in a DAA, if the maximal length of a route from a free node to N_i is len, the polynomial degree of N_i is $deg = (len + 1)(t - 1)$. The greatest polynomial degree of a node in a DAA is defined to be the* polynomial degree *of the DAA.*

Note that, an accumulating automaton with cycles (beyond self-cycles with corresponding character 1 as demonstrated in the sequel) implies an infinite polynomial degree.

Theorem 1. *For any DAA with polynomial degree d, we can implement and execute the DAA among d participants without communication and hide the (practically) unbounded input stream except* an upper bound on *the length of the input.*

Definition 6. *(*Accumulating field *of a DAA) The maximal number that should be represented by a marking variable in a dag accumulating automaton $\mathcal{DAA}$ is defined as* accumulating field *of $\mathcal{DAA}$.*

We should use sufficient *accumulating field* to avoid overflow during the execution. The total number of accumulating nodes $an \leq |V|$ and the maximal number of *active outgoing* edges $aoe \leq |V|$ of a node, imply a bound on the *accumulating field*. Each edge is active when the dealer assigns 1 to the label of the edge. Note that, unlike traditional deterministic automaton, in our case, there can be several edges from one node with the same label that lead to (at most $|V| - 1$) distinct nodes. Note that aoe is bounded by $|V| - 1$. The worst case is considered, where all accumulating nodes are lined one after the other (possibility according to a topological sort output), each multiplying its value by the number of outgoing arcs as an input to the next node in the line. Basically, for bounding the possible values, we consider the maximal value that can be accumulated in the i^{th} node to be the value that is added after multiplication by aoe, to the marking of the $(i+1)^{st}$ node with each input.

Theorem 2. *For an input stream of length n and a constant sized DAA the computing field of each node is in $\Theta(\log n)$ bits.*

5 The use of AA/DAA beyond string matching

In this section, we will give some applications of dag accumulating automaton which can recognize regular language, context free language and context sensitive language. We will also present several extensions to the transition function of directed accumulating automaton, namely: the possibility of the dealer to ignore characters, the possibility of loops with *unconditional arcs*, denoted by the label 1, and harvesting of result by comparing values. In some cases, the graph of the DAA is not connected allowing the implementation of every connected component by a different set of participants. We give the structure and initial marking of each DAA that can recognize a particular language in the above classes. Since every DAA can be securely and privately executed according to the scheme presented in Section 4, we only concern ourselves with the description of the DAA.

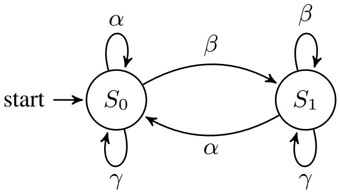


Fig. 8: Flip flop automaton $\mathcal{A}^{ff}$

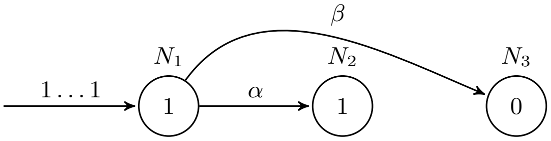


Fig. 9: $\mathcal{DAA}^{ff}$ for Implementing the flip flop automaton

5.1 Implementing flip flop automaton

Assume we have an automaton $\mathcal{A}^{ff}$ depicted in Fig 8 in which the initial state is S_0.
Initial marking and execution of $\mathcal{DAA}^{ff}$ Dag accumulating automaton $\mathcal{DAA}^{ff}$ of flip flop automaton can be found in Fig. 9. The alphabet of $\mathcal{DAA}^{ff}$ is $\Sigma = \{\alpha, \beta\}$. On initializing the automaton, N_1 is set to 1, N_2 is set to 1 and N_3 is set to 0. Let the $(k+1)^{th}$ input symbol be mapped to $\boldsymbol{v} = (v_0, v_1, v_2)$. The dealer will send different mapping vector depending on different input symbol. If the input symbol is α, $\boldsymbol{v}$ is set to $(1,1,0)$. If the input symbol is β, $\boldsymbol{v}$ is set to $(1,0,1)$. If the input symbol is γ, the dealer will *discard* it. Note that we allow such an action by the dealer, as well as sending spontaneous inputs and several characters in one input vector simultaneously. Then, we compute the new value of all the nodes as follows

$$\begin{array}{l} N_1^{(k+1)} = v_0 \\ N_2^{(k+1)} = N_1^{(k)} \cdot v_1 \\ N_3^{(k+1)} = N_1^{(k)} \cdot v_2 \end{array} \tag{5}$$

Result of $\mathcal{DAA}^{ff}$ After any input symbol, we can check the marking of $\mathcal{DAA}^{ff}$. If N_2 is 1, the current state of automaton $\mathcal{A}^{ff}$ is S_0. If N_3 is 1, the current state of automaton $\mathcal{A}^{ff}$ is S_1.
Correctness of $\mathcal{DAA}^{ff}$ According to the transitions of $\mathcal{DAA}^{ff}$, we can see (1) if the input symbol is α, N_2 will be set to 1; (2) if the input symbol is β, N_1 will be set to 0 and N_2 will be set to 1.

5.2 Recognizing regular language $(\alpha\beta\alpha)^*$ and $\alpha(\alpha\beta\alpha)^*$

Recognizing the regular language $(\alpha\beta\alpha)^*$
Dag accumulating automaton of the algorithm In Fig. 10 is the dag accumulating automaton $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$ for recognizing the regular language $(\alpha\beta\alpha)^*$. The alphabet of $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$ is $\Sigma = \{\alpha, \beta\}$. There is no free node in this automaton. Accumulating node is N_5.
Initial marking and execution of $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$ The first node N_1 is initially set to 1 while all the other nodes are initially set to 0. For each input symbol, we compute the new marking of the automaton. Let the $(k+1)^{th}$ input symbol be mapped to $\boldsymbol{v} = (v_1, v_2)$. If the input symbol is α, $\boldsymbol{v}$ is set to $(1,0)$. If the input symbol is β, $\boldsymbol{v}$ is set to $(0,1)$.

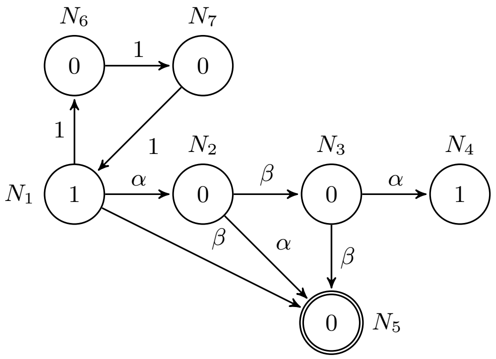


Fig. 10: $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$ for recognizing the regular language $(\alpha\beta\alpha)^*$ and the initial marking

We compute the new value of all the regular nodes as follows

$$\begin{aligned} N_1^{(k+1)} &= N_7^{(k)} \\ N_2^{(k+1)} &= N_1^{(k)} \cdot v_1 \\ N_3^{(k+1)} &= N_2^{(k)} \cdot v_2 \\ N_4^{(k+1)} &= N_3^{(k)} \cdot v_1 \\ N_6^{(k+1)} &= N_1^{(k)} \\ N_7^{(k+1)} &= N_6^{(k)} \end{aligned} \tag{6}$$

We compute the new value of accumulating node N_5 as follows

$$N_5^{(k+1)} = N_5^{(k)} + N_1^{(k)} \cdot v_2 + N_2^{(k)} \cdot v_1 + N_3^{(k)} \cdot v_2 \tag{7}$$

Result of $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$ After any input symbol, we can check the marking of $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$. Only if $N_4 = 1$ and $N_5 = 0$, the input stream is accepted, otherwise rejected.

Note, among the self-loop defined by N_1, N_6 and N_7, the degree for the secret sharing is not changed, since it involves multiplication by a constant 1.

Correctness of $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$ According to the transitions of $\mathcal{DAA}^{(\alpha\beta\alpha)^*}$, we can see (1) in the initial marking of the automaton, N_4 is set to 1, N_5 is set to 0; (2) if the input stream is $(\alpha\beta\alpha)^*$, N_4 will be set to 1, N_5 stay 0; (3) if the input stream is not $(\alpha\beta\alpha)^*$, N_4 will be set to 0 and/or N_5 will not be 0.

Recognizing the regular language $\alpha(\alpha\beta\alpha)^*$

Dag accumulating automaton of the algorithm In Fig. 11 is the dag accumulating automaton $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$ for recognizing the regular language $\alpha(\alpha\beta\alpha)^*$. The alphabet of $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$ is $\Sigma = \{\alpha, \beta\}$.

Initial marking and execution of $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$ The free node N_1 is initially set to 1 while all the other nodes are initially set to 0. For each input symbol, we compute the new marking of the automaton. Let the $(k+1)^{th}$ input symbol be mapped to $\boldsymbol{v} = (v_0, v_1, v_2)$ where v_0 is always set to 0. If the input symbol is α, $\boldsymbol{v}$ is set to $(0, 1, 0)$. If the input symbol is β, $\boldsymbol{v}$ is set to $(0, 0, 1)$.

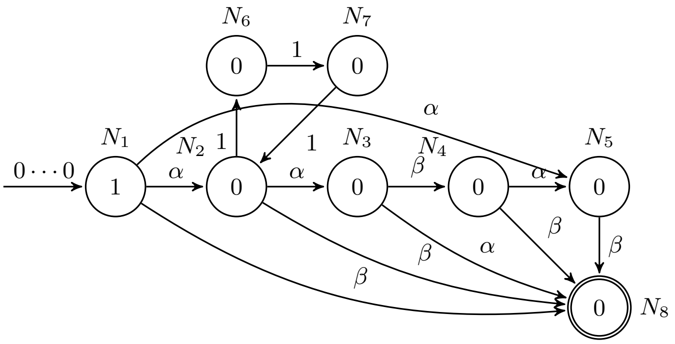


Fig. 11: $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$ for recognizing the regular language $\alpha(\alpha\beta\alpha)^*$ and the initial marking

We compute the new value of all the regular nodes as follows

$$\begin{aligned}
N_1^{(k+1)} &= v_0 \\
N_2^{(k+1)} &= N_1^{(k)} \cdot v_1 + N_7^{(k)} \\
N_3^{(k+1)} &= N_2^{(k)} \cdot v_1 \\
N_4^{(k+1)} &= N_3^{(k)} \cdot v_2 \\
N_5^{(k+1)} &= N_1^{(k)} \cdot v_1 + N_4^{(k)} \cdot v_1 \\
N_6^{(k+1)} &= N_2^{(k)} \\
N_7^{(k+1)} &= N_6^{(k)}
\end{aligned} \tag{8}$$

We compute the new value of the accumulating node N_8 as follows

$$\begin{aligned}
N_8^{(k+1)} = N_8^{(k)} &+ N_1^{(k)} \cdot v_2 + N_2^{(k)} \cdot v_2 \\
&+ N_3^{(k)} \cdot v_1 + N_4^{(k)} \cdot v_2 + N_5^{(k)} \cdot v_2
\end{aligned} \tag{9}$$

Result of $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$ After any input symbol, we can check the marking of $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$. Only if $N_5 = 1$ and $N_8 = 0$, the input stream is accepted, otherwise rejected.
Correctness of $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$ According to the transitions of $\mathcal{DAA}^{\alpha(\alpha\beta\alpha)^*}$, we can see (1) in the initial marking of the automaton, N_5 is set to 0, N_8 is set to 0; (2) if the input stream is α, N_5 will be set to 1, N_8 will stay 0; (3) if the input stream is $\alpha(\alpha\beta\alpha)^*$, N_5 will be set to 1, N_8 will stay 0; (4) if the input stream is not $\alpha(\alpha\beta\alpha)^*$ or α, N_5 will be set to 0 or N_8 will not equal 0.

5.3 Recognizing the context free language $\alpha^s\beta^s$

Dag accumulating automaton of the algorithm In Fig. 12 is the dag accumulating automaton $\mathcal{DAA}^{\alpha^s\beta^s}$ for recognizing the context free language $\alpha^s\beta^s$. The alphabet of $\mathcal{DAA}^{\alpha^s\beta^s}$ is $\Sigma = \{\alpha, \beta\}$.
Initial marking and execution of $\mathcal{DAA}^{\alpha^s\beta^s}$ All the free nodes N_1, N_3, N_5 are initially set to 1 while the other nodes are initially set to 0. Let the $(k+1)^{th}$ input symbol be mapped to $\boldsymbol{v} = (v_0, v_0', v_0'', v_1, v_2)$, where v_0, v_0' will always be set to 1 and v_0'' will always be set to 0. When we compute the new marking of the automaton, v_0 is given to N_1, v_0' is given to N_3 and v_0'' is given to N_5. If the input symbol is α, $\boldsymbol{v}$ is set to $(1, 1, 0, 1, 0)$. If the input symbol is β, $\boldsymbol{v}$ is set to $(1, 1, 0, 0, 1)$.

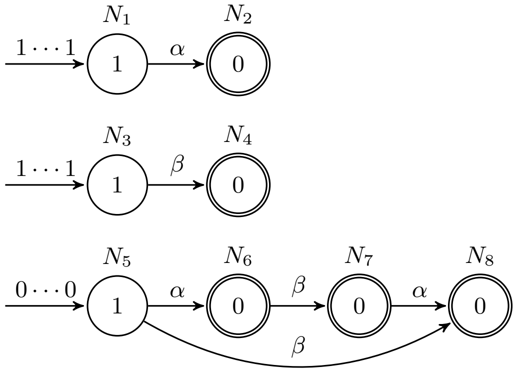


Fig. 12: $\mathcal{DAA}^{\alpha^s\beta^s}$ for recognizing the regular language $\alpha^s\beta^s$ and the initial marking

We compute the new value of all the regular nodes as follows

$$\begin{aligned} N_1^{(k+1)} &= v_0 \\ N_3^{(k+1)} &= v_1' \\ N_5^{(k+1)} &= v_1'' \end{aligned} \tag{10}$$

We compute all the accumulating nodes as follows

$$\begin{aligned} N_2^{(k+1)} &= N_2^{(k)} + N_1^{(k)} \cdot v_1 \\ N_4^{(k+1)} &= N_4^{(k)} + N_3^{(k)} \cdot v_2 \\ N_6^{(k+1)} &= N_6^{(k)} + N_5^{(k)} \cdot v_1 \\ N_7^{(k+1)} &= N_7^{(k)} + N_6^{(k)} \cdot v_2 \\ N_8^{(k+1)} &= N_8^{(k)} + N_7^{(k)} \cdot v_1 + N_5^{(k)} \cdot v_2 \end{aligned} \tag{11}$$

Result of $\mathcal{DAA}^{\alpha^s\beta^s}$ After any input symbol, we can check the marking of $\mathcal{DAA}^{\alpha^s\beta^s}$. If $N_8 > 0$, the input stream is rejected. Only if $N_2 = N_4$ and $N_8 = 0$, the input stream is accepted.
Correctness of $\mathcal{DAA}^{\alpha^s\beta^s}$ According to the transitions of $\mathcal{DAA}^{\alpha^s\beta^s}$ and the input mapping, node N_2 and N_4 count all the α and β symbols in the input stream respectively. While checking the input stream, if (1) the first input symbol is β, node N_8 is set to 1; (2) there is one or more α symbols after symbol β, node N_8 increases by 1.

5.4 Recognizing the context sensitive language $\alpha^s\beta^s\gamma^s$

Dag accumulating automaton of the algorithm In Fig. 13 is the dag accumulating automaton $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$ for recognizing the context sensitive language $\alpha^s\beta^s\gamma^s$. The alphabet of $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$ is $\Sigma = \{\alpha, \beta, \gamma\}$.

Initial marking and execution of $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$ All the free nodes $N_1, N_3,\ N_5, N_7$ are initially set to 1 while the other nodes are initially set to 0. Let the $(k+1)^{th}$ input symbol be mapped to $\boldsymbol{v} = (v_0, v_0', v_0'', v_0''', v_1, v_2, v_3)$, where v_0, v_0', v_0'' will always be set to 1 and v_0''' will always be set to 0. When we compute the new marking of the automaton, v_0 is given to N_1, v_0' is given to N_3, v_0'' is given to N_5 and v_0''' is given to N_7. If the input symbol is α, $\boldsymbol{v}$ is set to $(1,1,1,0,1,0,0)$. If the input symbol is β, $\boldsymbol{v}$ is set to $(1,1,1,0,0,1,0)$. If the input symbol is γ, $\boldsymbol{v}$ is set to $(1,1,1,0,0,0,1)$. The transitions are similar to the previous example, they are depicted in Fig. 13, and their detailed listing is omitted.

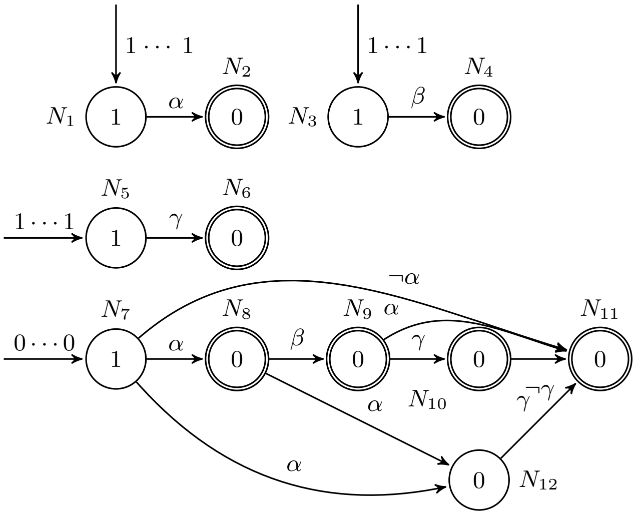


Fig. 13: $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$ for recognizing the regular language $\alpha^s\beta^s\gamma^s$ and the initial marking

Result of $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$ After any input symbol, we can check the marking of $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$. Only if $N_2 = N_4 = N_6$ and $N_{11} = 0$, the input stream is accepted, otherwise rejected.
Correctness of $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$ According to the transitions of $\mathcal{DAA}^{\alpha^s\beta^s\gamma^s}$ and the input mapping, node N_2, N_4 and N_6 count all the α, β and γ symbols in the input stream respectively. On checking the input stream, (1) if the first input symbol is not α, node N_{11} is set to 1; (2) if there is one α after symbol β, node N_{11} increases by 1; (3) if there is one γ after symbol α, node N_{11} increases by 1; (4) if there is one non-γ symbol after symbol γ, node N_{11} increases by 1.

6 Cascaded equations automata

In this section, we consider the same problem discussed in above sections under different scenario in which the input stream may be *actually* unbounded. We will solve this problem by introducing a new automata named cascaded equations automata. Firstly, we define what is cascaded equations and how to execute the cascaded equations. Then we show how to map the cascaded equations into automaton.

Definition 7 (Cascaded equations). *Cascaded equations is a series of equations ,* $e_1, e_2, \ldots, e_f$*, where the results and inputs of the first equations* $e_1, e_2, \ldots, e_i$ *are used to compute the result of the next equation* e_{i+1}.

An example of cascaded equations:

$$\begin{aligned} e_1 : N_1^{(k+1)} &= N_1^{(k)} + v_1 \\ e_2 : N_2^{(k+1)} &= N_2^{(k)} + N_1^{(k+1)} \cdot v_2 \\ e_3 : N_3^{(k+1)} &= N_3^{(k)} + N_1^{(k+1)} \cdot N_2^{(k+1)} \cdot v_3 \end{aligned} \tag{12}$$

In Equation 12, there are three equations e_1, e_2, e_3 with three variables N_1, N_2, N_3 and three inputs v_1, v_2, v_3.

Definition 8 (Execution of cascaded equations). *Cascaded equations are computed serially from e_1 to e_f. The first equation is computed, then the second and so on. At the end, the last equation is computed.*

Consider the following cascaded equations:

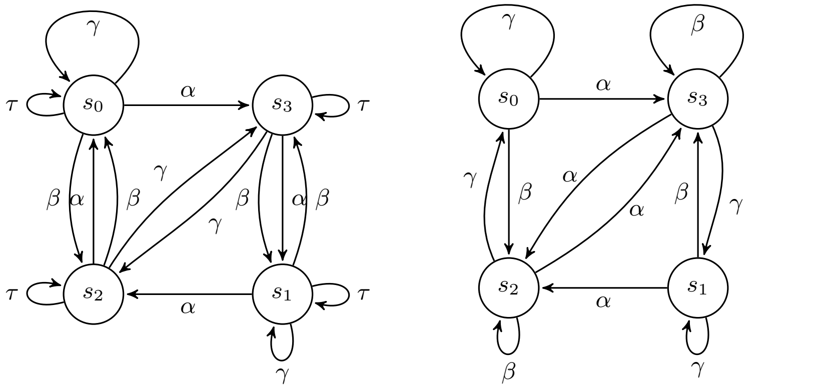
$$\begin{aligned} e_1 : N_1^{(k+1)} &= N_1^{(k)} + v_1 \\ e_2 : N_2^{(k+1)} &= N_2^{(k)} + N_1^{(k+1)} \cdot v_2 \end{aligned} \tag{13}$$

There are two equations, e_1 and e_2, in the cascaded equations described in Equation 13. The two equations compute a vector of variables (N_1, N_2) using a vector of inputs (v_1, v_2). Before executing the cascaded equations, we initialize the variables of the vector. Then, at the execution stage, we compute new values for N_1 and N_2 in a sequential fashion using modular two arithmetics, first using e_1 to compute N_1 and then e_2 that computes N_2. The input symbols (v_1, v_2) may have one of the possible values $(00), (01), (10)$ and (11). The state of the automaton is defined by a vector of the values of N_1 and N_2, this vector may have the following values $(00), (01), (10)$ or (11). A node of the automaton is denoted $s(N_1, N_2)$ and the input vector is denoted (11), (10), (01) and (00) by α, β, γ and τ respectively. By computing the cascaded equations using modular two arithmetics, we can obtain the automaton depicted in Fig. 14a.

Next, we consider the following cascaded equations:

$$\begin{aligned} e_1 : N_1^{(k+1)} &= v_1 \\ e_2 : N_2^{(k+1)} &= N_2^{(k)} + N_1^{(k+1)} \cdot v_2 \end{aligned} \tag{14}$$

By computing the cascaded equations in Equation 14 using arithmetic modular two, we can get the corresponding automaton which is depicted in Fig. 14b. A node is denoted $s_{(N_1 N_2)}$. The input vector (11), (10) and (01) are denoted by α, β and γ respectively.



(a) Cascaded permutation automata (b) Cascaded reset and permutation automata

Fig. 14: Automata mapped from two cascaded equations

Next, we explain multi-party execution of cascaded equations automata over strictly unbounded input stream. We use secret sharing to facilitate secure multi-party executions of the cascaded equations automata. The execution of cascaded equations automata is performed into three stages: initial stage, execution stage and collection stage. We use the automaton in Fig. 14a to demonstrate how the multi-party execution works.

Initial stage Each variable's values in the cascaded equation automata are shared among several participants using secret sharing. Entries of the vector that represent each symbol of the input symbol are also secret shared. For the particular example in Fig. 14a, we assume that the input symbols are represented by secret shares of polynomial of degree 1. If one equation includes multiplication, the degree of the polynomial that encodes the value of the variable will be more than the degree of the variable in the preceding equation. For the example in Fig. 14a, we have to use at least three participants to ensure that N_2 can be secret shared correctly among all the participants. For the two variable N_1 and N_2, we define two random polynomials f_1 and f_2 with degree 1 and 2. We use each corresponding polynomial to secret share each node's initial value among the three participants, each participant receives one share of N_1 and N_2.

Execution stage The dealer maps each input symbol α to an input vector $\boldsymbol{v}$. Then each element in the input vector $\boldsymbol{v}$ is secret shared into three parts by a random polynomial of degree 1. Each share of the input vector is then sent to one of the participants. Each participant computes the new value of N_1 and N_2, according to the Equation 13. Then, every participant gets the new share of N_1 and N_2.

Collection stage Whenever we want to compute the result of the algorithm, we ask all the participants to send the value that corresponds to N_1 and N_2 back. Having the shares of all participants, we can reconstruct the actual value of N_1 and N_2 using Lagrange interpolation. The value obtained indicates the current state of the automaton in Fig. 14a.

We give some formal definitions of cascaded equations automaton below.

Definition 9. *Mapping from cascaded equation to automaton Each equation in cascaded equations has a result. We select the results of the equations to define a vector. The vector's value encodes a node in the cascaded equations automaton. A vector of variables of the cascaded equations is regarded as the input symbols to the mapping automaton.*

Every cascaded equations can be mapped to an automaton by mapping variables of the equations into a node of the automaton.

Theorem 3. *The cascaded equations automata scheme information theoretically secures the inputs and the states of the automaton.*

We can also define a product of automata by executing several cascade automata in parallel. Two or more cascade equations with the same input can be merged together to obtain a new automaton.

Theorem 4. *Let $A = A_1 \times \ldots \times A_k$ be a cascade product of automata and let B be a permutation automaton. Let*

$$|A_1| = \ldots = |A_k| = |B|$$

and assume that for every $i = 1, ..., k$ the automaton A_i is either a reset automaton or a permutation automaton that can be represented by a cascaded equation, where all transitions are in the same cyclic group as the transitions of B. Then, A can be secretly shared for unbounded split input by $n+1$ parties with threshold 1 where n is computed as follows.

Computing n: Let Φ_i be a function of the input and the states of $A_1, ..., A_{i-1}$ that outputs the input for A_i. Representing Φ_i as a multivariate polynomial we have that its highest degree is of the form

$$x_1^{\alpha_1} \cdot \ldots \cdot x_{i_1}^{\alpha_{i-1}}$$

Define $n_i = n_1 \cdot \alpha_1 + ... + n_{i-1} \cdot \alpha_{i-1}$. Then, n is defined by $\max(n_1, ..., n_k)$.

This result can be further generalized by having B (and each A_i) be either a reset automaton or a set of non-intersecting permutation automata (i.e., there are several non-intersecting sets of nodes and each one is a permutation automaton). One additional generalization is the use of other modular operation (beyond mod 2) and hence larger fields. The realization of non-permutation automaton as in Fig. 14b, yields an important generalization of pure permutation automaton, since permutation automaton can be implemented by using only additions of secret shares.

7 Secure and private repeated computations on a secret shared file

In this section, we use methods introduced in the former sections on a fixed (large) file. Firstly, we secret share the file (e.g., biometric data) and store the shares in clouds for future computation. Then one can repeatedly and iteratively compute (for example, search the file for different strings) on the secret shared file by constructing the accumulating automaton for the needed computation and sending a copy of the automaton (possibly in different times) to each cloud that maintains shares of the file. Then, each cloud calculates on the accumulating automaton using their file share as the input. At the end, each cloud sends the final state of the accumulating automaton back as an answer for the computation request. The final states received from the cloud enable the reconstruction of the state of each node of the accumulating automaton to obtain the computation result (for example, whether the string was found or not). Sketch of the scheme is depicted in Fig. 15.

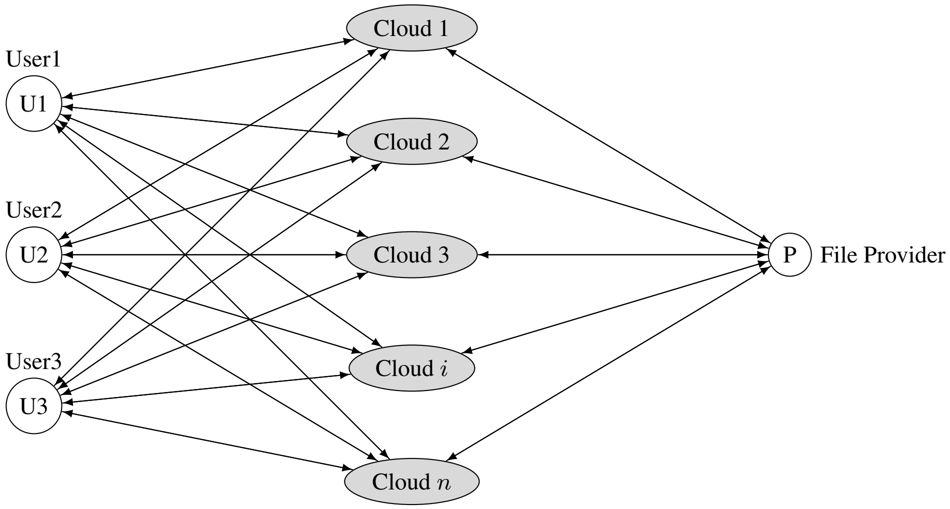


Fig. 15: Secure and private computation on a secret shared file among communicationless clouds

Next we detail the different stages of the scheme: $Setup$, $Initialization$, $AutomatonConstruction$, $AutomatonExection$ and $ResultReconstruction$.

Setup In this stage the basic parameters for the whole scheme are defined: the Alphabet (e.g., ASCII, binary) that the scheme works on, the computation field of all the accumulating automata and the biggest polynomial degree the system can deal with.

Initialization In this stage the given file f, the chosen Alphabet and number of clouds are used to output secret shares of each character of the file, where each character is encoded by a vector of secret shares, one secret share for each possible character.

AutomatonConstruction This stage uses the user computation task as an input and outputs an automaton.
AutomatonExecution This stage uses the accumulating automaton and the shares of the file to output the result of the computation. The result is the share of the final marking of the accumulating automaton.
ResultReconstruction This is the final stage in which the user receives the marking shares to output the computation result.

Next we demonstrate the scheme with a simple example. Assume provider Peter wants to store a network log file in clouds and user David wants to search the string "attack America" in the file. But Peter does not want to give the whole file to David in clear text. Firstly, Peter uses the *Initialization* stage to produce stream of shares of his log file (vector of shares for each character, character after a character) and then store each stream in a different cloud (or cloud virtual machine) not necessarily simultaneously. Clouds are not aware about their counterparts in the process. Then, David uses $AutomatonConstruction$ to get an accumulating automaton for the searching task (in some cases it is possible to give different independent parts of the accumulating automaton to different clouds). David sends the accumulating automaton to each cloud. Every cloud runs the $AutomatonExecution$ on their shares of the file and the accumulating automaton. Each cloud sends the marks of the final states of the accumulating automaton back to David. David executes $ResultReconstruction$ to find the computation result. During the whole procedure, no cloud knows the exact network log file and only David knows the computation result.

We have implemented a testing program of this system in SAGE, the source code can be found at: `http://blog.sina.com.cn/s/blog_9f0384e70101gj8x.html`.

8 Conclusion and discussion

In this paper, we proposed two new computation infrastructures the *accumulating automata* and the *cascaded equations automata* and their usage in the construction of secure and private multi-party computation among participants that use no communication among themselves while processing practically or really unbounded input stream. In particular, we can execute any string matching privately and securely in terms of information theoretically security. We demonstrate that other canonical examples of regular languages, context free languages and context sensitive languages can be computed efficiently in terms of information theoretical security. Remote authentication and data stream processing systems using cloud services can be implemented based on our schemes. We note that it is possible to design a general accumulating automata (in the style of FPGA) in which each original symbol is mapped to several symbols, so that the dealer is able to choose the non-participating arcs by always assigning zero to their labels. At last, the information sent by malfunctioning participants or even malicious participants may be eliminated from the collected information by standard (error correcting) schemes such as the Berlekamp Welch method [32].

References

1. Gilad Asharov and Yehuda Lindell. A full proof of the bgw protocol for perfectly-secure multiparty computation. *IACR Cryptology ePrint Archive*, 2011:136, 2011.
2. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.

3. Josh Cohen Benaloh. Secret sharing homomorphisms: Keeping shares of a secret sharing. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 1986.
4. John Bethencourt, Dawn Xiaodong Song, and Brent Waters. New techniques for private stream searching. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.
5. G. R. Blakley. Safeguarding cryptographic keys. *Managing Requirements Knowledge, International Workshop on*, 0:313, 1979.
6. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:111, 2011.
7. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.
8. Ivan Damgård and Rune Thorbek. Efficient conversion of secret-shared values between different fields. *IACR Cryptology ePrint Archive*, 2008:221, 2008.
9. Shlomi Dolev, Juan Garay, Niv Gilboa, and Vladimir Kolesnikov. Swarming secrets. In *Proceedings of the 47th annual Allerton conference on Communication, control, and computing*, Allerton'09, pages 1438–1445, Piscataway, NJ, USA, 2009. IEEE Press.
10. Shlomi Dolev, Juan A. Garay, Niv Gilboa, and Vladimir Kolesnikov. Secret sharing krohn-rhodes: Private and perennial distributed computation. In Bernard Chazelle, editor, *ICS*, pages 32–44. Tsinghua University Press, 2011.
11. Shlomi Dolev, Juan A. Garay, Niv Gilboa, Vladimir Kolesnikov, and Yelena Yuditsky. Brief announcement: Efficient private distributed computation on unbounded input streams. In Marcos K. Aguilera, editor, *DISC*, volume 7611 of *Lecture Notes in Computer Science*, pages 431–432. Springer, 2012.
12. Shlomi Dolev, Limor Lahiani, and Moti Yung. Secret swarm unit, reactive k-secret sharing. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2007.
13. Shlomi Dolev, Limor Lahiani, and Moti Yung. Secret swarm unit: Reactive k-secret sharing. *Ad Hoc Networks*, 10(7):1291–1305, 2012.
14. Wenliang Du and Mikhail J. Atallah. Secure multi-party computation problems and their applications: a review and open problems. In Victor Raskin, Steven J. Greenwald, Brenda Timmerman, and Darrell M. Kienzle, editors, *NSPW*, pages 13–22. ACM, 2001.
15. Christopher Fletcher, Marten van Dijk, and Srinivas Devadas. Compilation techniques for efficient encrypted computation. Cryptology ePrint Archive, Report 2012/266, 2012.
16. Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.
17. Craig Gentry. Toward basing fully homomorphic encryption on worst-case hardness. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 116–137. Springer, 2010.
18. Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2010:520, 2010.
19. Craig Gentry and Shai Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In Rafail Ostrovsky, editor, *FOCS*, pages 107–109. IEEE, 2011.
20. Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit. In Safavi-Naini and Canetti [27], pages 850–867.
21. Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
22. Shafi Goldwasser. Multi-party computations: Past and present. In James E. Burns and Hagit Attiya, editors, *PODC*, pages 1–6. ACM, 1997.
23. Yan Huang, Jonathan Katz, and David Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy*. IEEE, 2012.
24. Yan Huang, Jonathan Katz, and David Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology–CRYPTO 2013*, pages 18–35. Springer, 2013.
25. Payman Mohassel, Salman Niksefat, Seyed Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 398–415. Springer, 2012.
26. Rafail Ostrovsky and William E Skeith III. Private searching on streaming data. *Journal of Cryptology*, 20(4):397–430, 2007.
27. Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.
28. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

29. abhi shelat and Chih-hao Shen. Fast two-party secure computation with minimal assumptions. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 523–534, New York, NY, USA, 2013. ACM.
30. Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.
31. Brent Waters. Functional encryption for regular languages. In Safavi-Naini and Canetti [27], pages 218–235.
32. Lloyd R Welch and Elwyn R Berlekamp. Error correction for algebraic block codes, December 30 1986. US Patent 4,633,470.