

# Toward practical encrypted email that supports private, regular-expression searches

Lei Wei · Michael K. Reiter

Published online: 22 November 2014  
© The Author(s) 2014. This article is published with open access at Springerlink.com

**Abstract** In this paper, we develop a protocol to enable private regular-expression searches on encrypted data stored at a **server**. A novelty of the protocol lies in allowing a user to securely delegate an encrypted search query to a **proxy**, which interacts with the **server** where the user’s data are stored encrypted to produce the search result for the user. The privacy of the query and the data are both provably protected against an arbitrarily malicious **server** and an honest-but-curious **proxy** under rigorous security definitions. We then detail a series of optimizations to our initial design that achieve an order-of-magnitude performance improvement over the original protocol. We demonstrate the practicality of the resulting protocol through measurements of private regular-expression searches on a real-world email dataset.

**Keywords** Private search on encrypted data · Regular-expression search · Cloud security

## 1 Introduction

The tension between encryption and the ability to perform searches on the underlying plaintext has been a topic of considerable attention in the research community. Numerous designs of so-called *searchable encryption* schemes have been proposed to address this tension (see Sect. 2 for a

detailed discussion), but few have made it to practical use. We believe that this state of affairs is in part due to inflexibility, in the sense that such schemes typically require the document creator to tag it by the keywords on which searches will be supported in the future. Even though one can imagine tagging and encrypting all the words in a document to allow for all searches on any word, anything from a typo in the document to different forms of word stemming will render the search results unsatisfactory. For example, a document tagged with the keyword “meetings” would not be returned in the search results for the queries “meet” or “meeting”. A recent study of user email query patterns [1] showed that many queries that users create are only partial words, and so, substring searching capability is important to provide an adequate user experience. Furthermore, very few searchable encryption schemes offer the capabilities of performing substring, conjunctive, disjunctive, and range queries, and we are aware of none that offers all them at the same time.

In this paper, we develop a protocol to enable private regular-expression searches on encrypted data and a system that demonstrates this capability on encrypted email. Regular-expression searches are a widely adopted search primitive in many languages and programming frameworks<sup>1</sup> (e.g., see [3]) and, for example, suffice to support the search options (including Boolean combinations) offered by the Thunderbird email client for the text and numeric email fields. Our system is thus able to support range queries on the date field and various types of substring queries on the source, destination and subject fields of emails.

---

L. Wei · M. K. Reiter (✉)  
Department of Computer Science, University of North Carolina  
at Chapel Hill, Chapel Hill, NC, USA  
e-mail: reiter@cs.unc.edu

L. Wei  
e-mail: lwei@cs.unc.edu

*Present Address:*

L. Wei  
Apple, Inc., Cupertino, CA, USA

---

<sup>1</sup> To be more precise, the term “regular expression” is used in some frameworks in a way that follows but deviates somewhat from its original definition. Our system supports searches using regular expressions as originally defined, i.e., searches that can be expressed as deterministic finite automata [2].

Our protocol for regular-expression searching gains computational efficiency by using interaction, in fact requiring data transfer between the searching client and the **server** holding the ciphertext of a volume larger than the searchable ciphertext itself. This obviously begs the question of whether a more suitable solution would be to download each email to the client and decrypt it there, to be searched locally. With the widespread use of volume-priced networking (i.e., over cellular data plans), however, neither design is particularly appealing. So, we instead explore a different design in which the **user** (e.g., via her mobile device) submits her *encrypted* regular expression (or suitable representation thereof) to a **proxy**, which then interacts with the **server** hosting the encrypted data using our protocol. After this interaction, the **proxy** reports information back to the **user** that permits her to determine which files matched, so she can retrieve these files from the **server**. We stress that the volume of interaction between the **user** and the **proxy** is independent of the lengths of the ciphertexts stored at the **server**. Moreover, the **proxy** is untrusted for the privacy of the search or the file contents (provided that it does not collaborate maliciously with the **server**), provably so when the **proxy** is honest-but-curious. So, for example, the **proxy** could be a machine that is well defended and closely monitored to ensure its integrity, but nevertheless untrusted with the details of the **user**'s searches or the emails they match—as might be the case with a **user**'s professionally maintained work computer, through which she also searches her personal email.

The task of constructing such a protocol to be efficient is, as we found, very challenging. Starting from a protocol that implements the above functionality, we detail a series of optimizations that resulted in an optimized protocol with more than an order-of-magnitude improvement in the performance, while enjoying similar security properties. At a high level, the optimizations involve careful redesign of the protocol in order to take advantage of well-known algebraic optimization techniques (e.g., preprocessing to optimize pairing operations) and a few novel algebraic techniques to reduce the online computational costs. After detailing these protocol optimizations, we then explore additional optimizations that leverage specifics of the email setting. These optimizations pertain to the specific regular-expression alphabet that should be utilized for each type of searchable field (i.e., source email address, sender name, date, and subject).

We detail an implementation of our protocol and its performance when searching emails from a real-world email dataset. We show, for example, that our implementation incurs average latencies of 0.89 s per email for performing a nine-character substring search on the sender email address field and 0.17 s per email for performing a range query spanning about six months on the email date field (These numbers were obtained from a **proxy** and **server** each having eight 2.67 GHz cores with simultaneous multithreading enabled).

We also evaluate options for exploiting parallelism with our protocol, ranging from very coarse (i.e., one **server** thread and one **proxy** thread per **server-proxy** protocol instance, but running many protocols instances in parallel) to very fine (i.e., many **server** threads and **proxy** threads in one protocol instance). While our protocol admittedly does not offer sufficient user responsiveness to search the many thousands of emails in our own email folders while we wait, it is easily efficient enough to search the small minority of those messages that are encrypted or, we believe, that would merit encryption. Moreover, we anticipate other usage modes for which our protocol's search performance should be more than sufficient: e.g., a **user** can register a long-standing “subscription” query at the **proxy**, which can evaluate the query on each email message as it arrives at the **server** and inform the **user** of the result.

To summarize, our contributions are as follows. First, we provide a design for an interactive protocol by which (i) a **user** provides an encrypted representation of a regular expression to a **proxy**, (ii) the **proxy** interacts with a **server** holding an encrypted file to produce an output for the **proxy**, and (iii) upon receiving that output from the **proxy**, the **user** can determine whether the regular expression matched that file. However, neither the **proxy** nor the **server** learns anything about the search result, the regular expression (except its size), or the file plaintext (except its size). Second, we develop an implementation of this protocol together with several optimizations to make it perform well and then evaluate the performance of this implementation on a real-world email dataset.

## 2 Related work

The problem we study falls into the general area of secure “computing on encrypted data” [4] or two-party secure computation [5,6] and could be implemented using general techniques. The former achieves computations noninteractively using fully homomorphic encryption, for which existing implementations (e.g., [7–10]) are still far from practical. The latter utilizes “garbled circuits” of size linear in the circuit representation of the function to be computed. Despite progress on practical implementations of this technique [11–14], this limitation renders it much more computation and communication intensive for the problem we consider. Perhaps, more importantly, since our protocol aims to enable a resource-constrained **user** to outsource the search query and workload to a **proxy**, the interaction between the **user** and **proxy** should be minimized. Using our protocol, the communication cost in the direction from the **user** to the **proxy** is only dependent on the size of the search query and is independent of the number and size of the file ciphertexts. We are unaware of how to achieve this property using garbled

circuits, however. Since the garbled circuit and its inputs are “unreusable” across different runs of the protocol, the user would need to provide a number of inputs (in this case, encrypted queries) to the **proxy** that equals to the number of files to be searched. Furthermore, the fact that in our construction, the user-generated encrypted query can be used an unlimited number of times enables a “subscription” service in which the **proxy** holds the encrypted query and periodically informs the user of the arrival of matched emails, without any further communication from the user to the **proxy**. Again, we are unaware of how to implement this functionality using generic garbled circuit techniques.

The protocol we design here directly builds on a previous protocol [15] that enables a client to evaluate a deterministic finite automaton (DFA) on the plaintext of an encrypted file hosted by a server provided that the client has been given permission to do so by the data owner. Our protocol differs by permitting the client (in our case, the **proxy**) to perform this evaluation using an *encrypted* DFA provided by the **user**, so that the **proxy** need not be trusted with the privacy of the search query. We also contribute over that prior work by developing substantial optimizations to the performance of the resulting protocol; many of these optimizations should be applicable to the previous protocol, as well.

The structure of our protocol, involving a **user** who splits the DFA evaluation between a **proxy** and **server**, is reminiscent of a protocol due to Blanton and Aliasgari [16]. Their protocol secret-shares the DFA and (plaintext) file, respectively, between two hosts, which then interactively evaluate the DFA on the file without reconstructing either one. Their protocol, however, does not support asymmetric encryption of the file, which in the email context that our protocol is designed to support, is the predominant method for preparing a private email for its intended recipient. With our protocol, as with today’s encrypted email options, the email sender needs only a public key for the intended recipient.

There has been a significant work on searchable symmetric encryption (SSE) (e.g., [17–24]), the key for which an email sender could encrypt using the email recipient’s public key after using it to encrypt the searchable email fields. This would make the search functionality provided by the SSE scheme available to the **user**, but only after the **user** downloads the encrypted symmetric key for each email, so that she can create search queries for each one. In contrast, the protocol we present here does not require the **user** to obtain per email information before creating a search query for all emails. As mentioned above, the **proxy** could even hold and apply a “subscription” query periodically to inform the **user** of newly arrived emails that match it. Using searchable asymmetric encryption (e.g., [19,22,23,25–27]) to encrypt each email could provide this feature, though we are unaware of any such scheme that supports regular-expression queries.

Because our protocol between the **proxy** and **server** is interactive, other interactive protocols are also related to our work. (Searchable encryption schemes are typically noninteractive.) In particular, interactive protocols for two-party private DFA evaluation, in which a **server** has a file and a **user** has a DFA to evaluate on that file, have been a topic of recent focus (e.g., [28–31]). Our work differs from these in that in our protocols, the file is available to the parties only in ciphertext form. We overcome this obstacle by the **user** two-party sharing the email-decryption key between the **proxy** and the **server**, so that they can interact to evaluate the **user**’s DFA on encrypted email fields. A protocol due to Choi et al. [32] takes a similar approach to enable general two-party computations on data that resides only in encrypted form at the parties. However, because this protocol is based on garbled circuits, it inherits the drawbacks we mentioned earlier.

Returning to the noninteractive case, Waters [33] described a functional encryption scheme that allows a secret key tied to a specific DFA to decrypt a ciphertext if the DFA accepts a fixed string associated with the ciphertext. However, his security requirements are quite different from ours, in that the string to be matched is considered public in his context. In our case, this string corresponds to the encrypted file and so must remain private.

### 3 Protocol design

We describe our protocols for regular-expression evaluation by assuming the regular expression has first been translated to an equivalent deterministic finite automaton (DFA) [2]. A DFA  $M$  is a tuple  $\langle Q, \Sigma, \delta, q_{\text{init}}, \Delta \rangle$  where  $Q$  is a set of  $|Q| = n$  states;  $\Sigma$  is a set (*alphabet*) of  $|\Sigma| = m$  symbols;  $\delta : Q \times \Sigma \rightarrow Q$  is a transition function;  $q_{\text{init}}$  is the initial state; and  $\Delta : Q \rightarrow \{0, 1\}$  is a function for which  $\Delta(q) = 1$  indicates that  $q$  is an accepting state.

#### 3.1 Our starting point

We first review the protocol that serves as our starting point [15]. The protocol enables a **client** having a DFA  $M$  to interact with a **server** storing the ciphertext of a file to obtain the result as though  $M$  was evaluated on the file plaintext. More precisely, the **client** should output a bit indicating whether the final state to which the file plaintext drives the DFA is accepting or not; i.e., if the plaintext of the file is a sequence  $\langle \sigma_k \rangle_{k \in [\ell]}$  where  $[\ell]$  denotes the set  $\{0, 1, \dots, \ell - 1\}$  and where each  $\sigma_k \in \Sigma$ , then the **client** should output  $\Delta(\delta(\dots \delta(\delta(q_{\text{init}}, \sigma_0), \sigma_1), \dots, \sigma_{\ell-1}))$ . The **client** can learn the file length  $\ell$ , and the **server** can learn both  $\ell$  and the number of states  $n$  in the client’s DFA. The **client** should learn

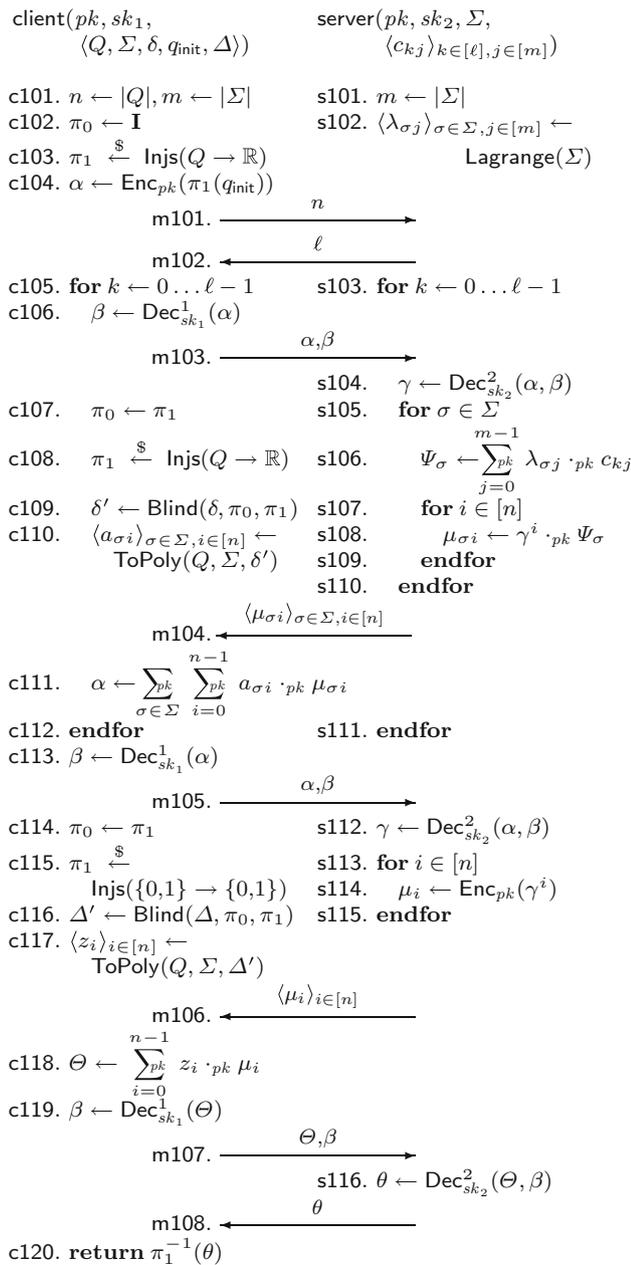


Fig. 1 Protocol  $\Pi_1(\mathcal{E})$

nothing else about the file; however, the server should learn nothing else about the file or the client’s DFA.

The protocol is shown in Fig. 1. The protocol is built using an additively homomorphic encryption scheme with plaintext space  $\mathbb{R}$  where  $(\mathbb{R}, +_{\mathbb{R}}, \cdot_{\mathbb{R}})$  denotes a commutative ring. Specifically, an encryption scheme  $\mathcal{E}$  includes algorithms Gen, Enc, and Dec where: Gen is a randomized algorithm that on input  $1^k$  outputs a public key/private key pair  $(pk, sk) \leftarrow \text{Gen}(1^k)$ ; Enc is a randomized algorithm that on input public key  $pk$  and plaintext  $m \in \mathbb{R}$  (where  $\mathbb{R}$  can be determined as a function of  $pk$ ) produces a ciphertext  $c \leftarrow \text{Enc}_{pk}(m)$ , where  $c \in C_{pk}$ ;  $C_{pk}$  is the ciphertext space

determined by  $pk$ ; and Dec is a deterministic algorithm that on input a private key  $sk$  and ciphertext  $c \in C_{pk}$  produces a plaintext  $m \leftarrow \text{Dec}_{sk}(c)$  where  $m \in \mathbb{R}$ . In addition,  $\mathcal{E}$  supports an operation  $+_{pk}$  on ciphertexts such that for any public key/private key pair  $(pk, sk)$ ,  $\text{Dec}_{sk}(\text{Enc}_{pk}(m_1) +_{pk} \text{Enc}_{pk}(m_2)) = m_1 +_{\mathbb{R}} m_2$ . Using  $+_{pk}$ , it is possible to implement  $\cdot_{pk}$  for which  $\text{Dec}_{sk}(m_2 \cdot_{pk} \text{Enc}_{pk}(m_1)) = m_1 \cdot_{\mathbb{R}} m_2$ . We use  $\sum_k$  to denote summation using  $+_{pk}$ ;  $\sum_{\mathbb{R}}$  to denote summation using  $+_{\mathbb{R}}$ ; and  $\prod$  to denote the product using  $\cdot_{\mathbb{R}}$  of a sequence.

The protocol also requires  $\mathcal{E}$  to support two-party decryption. Specifically, there exists an efficient randomized algorithm Share that on input a private key  $sk$  outputs shares  $(sk_1, sk_2) \leftarrow \text{Share}(sk)$ , and there are efficient deterministic algorithms  $\text{Dec}^1$  and  $\text{Dec}^2$  such that  $\text{Dec}_{sk}(c) = \text{Dec}_{sk_2}^2(c, \text{Dec}_{sk_1}^1(c))$ . An example of an encryption scheme  $\mathcal{E}$  that meets the above requirements is due to Paillier [34] with modifications by Damgård and Jurik [35].

The main ingredient of the protocol is a method to encode a DFA  $(Q, \Sigma, \delta, q_{\text{init}}, \Delta)$ , and specifically the transition function  $\delta$ , as a bivariate polynomial  $f(x, y)$  over  $\mathbb{R}$  where  $x$  is the variable representing a DFA state and  $y$  is the variable representing an input symbol. That is, if we treat each state  $q \in Q$  and each  $\sigma \in \Sigma$  as distinct elements of  $\mathbb{R}$ , then we would like  $f(q, \sigma) = \delta(q, \sigma)$ . This can be achieved by choosing  $f$  to be the interpolation polynomial

$$f(x, y) = \sum_{\sigma \in \Sigma} (f_{\sigma}(x) \cdot_{\mathbb{R}} \Lambda_{\sigma}(y)) \tag{1}$$

where  $f_{\sigma}(q) = \delta(q, \sigma)$  for each  $q \in Q$  and where

$$\Lambda_{\sigma}(y) = \prod_{\substack{\sigma' \in \Sigma \\ \sigma' \neq \sigma}} \frac{y -_{\mathbb{R}} \sigma'}{\sigma -_{\mathbb{R}} \sigma'} \tag{2}$$

is a Lagrange basis polynomial. Note that  $\Lambda_{\sigma}(\sigma) = 1$  and  $\Lambda_{\sigma}(\sigma') = 0$  for any  $\sigma' \in \Sigma \setminus \{\sigma\}$ .

Calculating (2) requires taking multiplicative inverses in  $\mathbb{R}$ . While not every element of a ring has a multiplicative inverse in the ring, fortunately, the ring used in Paillier encryption, for example, has negligibly few elements with no inverses, and so, there is little risk of encountering an element with no inverse. Using Eq. 2, we can calculate coefficients  $\langle \lambda_{\sigma j} \rangle_{j \in [m]}$ , so that

$$\Lambda_{\sigma}(y) = \sum_{j=0}^{m-1} \lambda_{\sigma j} \cdot_{\mathbb{R}} y^j$$

This calculation is encapsulated in the procedure  $\langle \lambda_{\sigma j} \rangle_{\sigma \in \Sigma, j \in [m]} \leftarrow \text{Lagrange}(\Sigma)$  in Fig. 1.

Each  $f_{\sigma}$  needed to compute Eq. 1 can again be determined as an interpolation polynomial in the Lagrange form and then expressed as  $f_{\sigma}(x) = \sum_{i=0}^{n-1} a_{\sigma i} \cdot_{\mathbb{R}} x^i$ . In Fig. 1, this

calculation is represented by the operation  $\langle a_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]} \leftarrow \text{ToPoly}(Q, \Sigma, \delta)$ .

At the beginning of the protocol, the client receives as input a public key  $pk$  under which the file stored at the server is encrypted. In addition, he receives a share  $sk_1$  of the private key  $sk$  corresponding to  $pk$ , and the DFA  $\langle Q, \Sigma, \delta, q_{\text{init}}, \Delta \rangle$ . The server receives as input the public key  $pk$ , another share  $sk_2$  of the private key  $sk$ , the DFA alphabet  $\Sigma$ , and the file ciphertexts  $\langle c_{kj} \rangle_{k \in [\ell], j \in [m]}$  where  $c_{kj} \leftarrow \text{Enc}_{pk}((\sigma_k)^j)$ .

The protocol proceeds in rounds, processing one character in each round. The client begins the  $k$ th loop iteration with an encryption  $\alpha$  of the current DFA state after being blinded by a random injection  $\pi_1 : Q \rightarrow \mathbb{R}$  it chose in the  $(k - 1)$ th loop at line c108 (or, if  $k = 0$ , then in line c103), where  $\text{Injs}(Q \rightarrow \mathbb{R})$  denotes the set of injections from  $Q$  to  $\mathbb{R}$  ( $\mathbf{I}$  denotes the identity function in c102). The client uses its share  $sk_1$  to partially decrypt  $\alpha$  (c106) and sends the result to the server. The server uses his share  $sk_2$  to fully decrypt  $\alpha$  and obtains the blinded state  $\gamma$  (s104). The server then computes, for each  $\sigma \in \Sigma$  (s105), a value  $\Psi_\sigma$  such that  $\Lambda_\sigma(\sigma_k) = \text{Dec}_{sk}(\Psi_\sigma)$  (s106) by utilizing coefficients  $\langle \lambda_{\sigma j} \rangle_{\sigma \in \Sigma, j \in [m]}$  output from Lagrange (s102). The server then returns (in s104) values  $\langle \mu_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$  created, so that  $\text{Dec}_{sk}(\mu_{\sigma i}) = \gamma^i \cdot_{\mathbb{R}} \Lambda_\sigma(\sigma_k)$  (s108).

Meanwhile, the client selects a new random injection  $\pi_1 \xleftarrow{\$} \text{Injs}(Q \rightarrow \mathbb{R})$  (c108) and constructs a new DFA transition function  $\delta'$  reflecting the injection it chose in the last round (now denoted  $\pi_0$ , see line c107) and the new injection  $\pi_1$  it chose for this round. Specifically, it creates a new DFA state transition function  $\delta'$  defined as  $\delta'(q, \sigma) = \pi_1(\delta(\pi_0^{-1}(q), \sigma))$  for all  $\sigma \in \Sigma$  and  $q \in \pi_0(Q)$  where  $\pi_0(Q) = \{\pi_0(q)\}_{q \in Q}$ . This step is denoted as  $\delta' \leftarrow \text{Blind}(\delta, \pi_0, \pi_1)$  in line c109. That is,  $\delta'$  “undoes” the previous injection  $\pi_0$ , applies  $\delta$ , and then applies the new injection  $\pi_1$ . The client then interpolates a new bivariate polynomial  $f(x, y)$  such that  $f(q, \sigma) = \delta'(q, \sigma)$  in line c110, using the algorithm described previously. The client uses these coefficients and  $\langle \mu_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$  sent from the server (m103) to “evaluate” the polynomial and obtains a ciphertext  $\alpha$  of the new DFA state under the injection  $\pi_1$  (c111). After  $\ell$  loop iterations, the client obtains the encrypted final state in c111.

The client then engages in another round of interaction with the server in order to obtain a binary answer of whether the final state is an accepting state or not. For that purpose, the client creates another polynomial  $F(x) = \sum_{i=0}^{n-1} z_i \cdot_{\mathbb{R}} x^i$  such that  $F(q) = 1$  if and only if  $\Delta(q) = 1$  and  $F(q) = 0$  otherwise (c117). That is,  $F(x)$  “converges” all accepting states to 1 and all nonaccepting states to 0. Since the client needs help from the server to decrypt the final result (s116), it applies another random injection  $\pi_1 \xleftarrow{\$} \text{Injs}(\{0, 1\} \rightarrow \{0, 1\})$  on the output of the function  $\Delta$  to hide the results from the server. In the protocol, we use  $\Delta' \leftarrow \text{Blind}(\Delta, \pi_0, \pi_1)$

(c116) to denote the step to generate the blinded function that maps the accepting state to a random element of  $\{0, 1\}$ . The client then uses the polynomial interpolation procedure to obtain the coefficients of  $F(x)$  in c117. After “evaluating”  $F(x)$  in c118 and obtaining the encrypted binary output  $\Theta$ , the client interacts with the server once more to decrypt it and returns the result in c120.

### 3.2 Our initial construction

Starting from the protocol of the previous section, in this section, we develop the first contribution of this paper, namely a protocol that replaces the client with two parties: a user that holds the DFA  $\langle Q, \Sigma, \delta, q_{\text{init}}, \Delta \rangle$  and a proxy that the user invokes to conduct a protocol to evaluate this DFA on a file stored at the server. Notably, the protocol we develop here protects the secrecy of the DFA  $\langle Q, \Sigma, \delta, q_{\text{init}}, \Delta \rangle$  and the evaluation result from the proxy, and so, this modification enables the proxy to execute the protocol on behalf of others who do not trust it with knowledge of the DFA. One scenario in which this protection is desirable is if the user does not have the bandwidth or processing available for performing the evaluation herself.

The protocol, denoted  $\Pi_2(\mathcal{E})$ , protects the DFA privacy by giving to the proxy the encryptions of the coefficients of the DFA polynomial  $f$ , denoted  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$  where  $\hat{a}_{\sigma i} \leftarrow \text{Enc}_{pk}(a_{\sigma i})$  and  $\langle a_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]} \leftarrow \text{ToPoly}(Q, \Sigma, \delta)$ , and the encryptions of the coefficients of the converging polynomial  $F$ , i.e.,  $\langle \hat{z}_i \rangle_{i \in [n]}$  where  $\hat{z}_i \leftarrow \text{Enc}_{pk}(z_i)$  and  $\langle z_i \rangle_{i \in [n]} \leftarrow \text{ToPoly}(Q, \Sigma, \Delta)$ . (The values  $\langle z_i \rangle_{i \in [n]}$  do not depend on  $\Sigma$ , but we continue to provide  $\Sigma$  to  $\text{ToPoly}$  for consistency.) The implications of this change to the protocol are far reaching, due to the operations that the proxy needs to perform using these now-encrypted coefficients.

In the original protocol, in order to hide the current state transition from the server, the client blinds the current transition state by choosing a random injection  $\pi_1$  of the state encodings in each round, so that the server obtains a random ring element  $\gamma$  in s104 every time. A new DFA polynomial is then interpolated to accommodate the injections chosen in the last and current round (c107–c110) to continue state transitions consistently. When the coefficients are encrypted, however, the proxy will not be able to interpolate new polynomials because it does not have access to  $\delta$ . We thus need another strategy to achieve these “blinding” and “unblinding” effects. Rather than blinding with a random injection, the new protocol does so by adding in a random ring element  $r$  to the ciphertext representing the current state (c203–c204). The consequence of this additive blinding operation is that the proxy needs a way to “shift”  $f$  (and its encrypted coefficients) to produce a polynomial  $f'(x, y)$  satisfying  $f'(q +_{\mathbb{R}} r, \sigma) = \delta(q, \sigma)$  for each  $q \in Q$  and  $\sigma \in \Sigma$ , for a specified  $r \in \mathbb{R}$ . If we set

$$f'(x, y) = \sum_{\sigma \in \Sigma} (f'_\sigma(x) \cdot_{\mathbb{R}} \Lambda_\sigma(y))$$

where  $f'_\sigma(x) = \sum_{i=0}^{n-1} a'_{\sigma i} \cdot_{\mathbb{R}} x^i$ , then it suffices if  $f'_\sigma(x +_{\mathbb{R}} r) = f'_\sigma(x)$  for all  $\sigma \in \Sigma$ . Note that

$$\begin{aligned} f'_\sigma(x -_{\mathbb{R}} r) &= \sum_{i=0}^{n-1} a_{\sigma i} \cdot_{\mathbb{R}} (x -_{\mathbb{R}} r)^i \\ &= \sum_{i=0}^{n-1} a_{\sigma i} \cdot_{\mathbb{R}} \sum_{i'=0}^i \binom{i}{i'} \cdot_{\mathbb{R}} x^{i-i'} \cdot_{\mathbb{R}} (-_{\mathbb{R}} r)^{i'} \\ &= \sum_{i=0}^{n-1} \left( \sum_{i'=0}^{n-1-i} a_{\sigma(i+i')} \cdot_{\mathbb{R}} \binom{i+i'}{i'} \cdot_{\mathbb{R}} (-_{\mathbb{R}} r)^{i'} \right) \cdot_{\mathbb{R}} x^i \end{aligned} \tag{3}$$

where Eq. 3 follows from the binomial theorem. As such, setting

$$a'_{\sigma i} \leftarrow \sum_{i'=0}^{n-1-i} a_{\sigma(i+i')} \cdot_{\mathbb{R}} \binom{i+i'}{i'} \cdot_{\mathbb{R}} (-_{\mathbb{R}} r)^{i'} \tag{4}$$

ensures  $f'_\sigma(x +_{\mathbb{R}} r) = f'_\sigma(x)$  and, therefore,  $f(x +_{\mathbb{R}} r, \sigma) = f'(x, \sigma)$ . When the proxy has access to only the encrypted coefficients, represented by  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$ , the operation in Eq. 4 needs to be changed to

$$\hat{a}'_{\sigma i} \leftarrow \sum_{i'=0}^{n-1-i} \left( \binom{i+i'}{i'} \cdot_{\mathbb{R}} (-_{\mathbb{R}} r)^{i'} \right) \cdot_{pk} \hat{a}_{\sigma(i+i')} \tag{5}$$

In our pseudocode, we encapsulate calculations of Eq. 5 in the invocation  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]} \leftarrow \text{Shift}(r, \langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]})$ .

Now that the coefficients are encrypted, the operation by the client to combine coefficients with ciphertexts as was done in line c111 in Fig. 1 no longer works for the proxy. For this reason, we need to expand the properties we require of the encryption system we use, to include the ability to homomorphically “multiply” ciphertexts *once*. We emphasize that we do *not* require fully homomorphic encryption. Our construction can be instantiated with any additively homomorphic encryption scheme that allows a single homomorphic multiplication of two ciphertexts (e.g., [36,37]), provided that it also supports two-party decryption. Here, we build from the more well-studied scheme of Boneh et al. [36], which we denote by BGN.<sup>2</sup>

**Encryption scheme** BGN uses an algorithm BGNInit that, on input  $1^\kappa$ , outputs  $(p, p', \mathbb{G}, \mathbb{G}', e)$  where  $p, p'$  are random  $\kappa/2$ -bit primes,  $\mathbb{G}$  and  $\mathbb{G}'$  are cyclic groups of order

<sup>2</sup> Our previous work [15] includes a protocol that also leverages BGN but for a completely different purpose, namely simply improving the communication complexity between the client and server.

$N = pp'$ , and  $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}'$  is a bilinear map. In this encryption scheme, the ring  $\mathbb{R}$  is  $\mathbb{Z}_N$ , the ciphertext space  $C_{(N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g})}$  is  $\mathbb{G} \cup \mathbb{G}'$ , and the relevant algorithms are defined as follows. Note that we assume that elements of  $\mathbb{G}$  and  $\mathbb{G}'$  are encoded distinctly.

**Gen**( $1^\kappa$ ): Generate  $(p, p', \mathbb{G}, \mathbb{G}', e) \leftarrow \text{BGNInit}(1^\kappa)$ ; select random generators  $g, u \xleftarrow{\$} \mathbb{G}$ ; set  $N \leftarrow pp', h \leftarrow u^{p'}$ , and  $\hat{g} \leftarrow e(g, g)^p$ ; and return public key  $\langle N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g} \rangle$  and private key  $\langle N, \mathbb{G}, \mathbb{G}', e, g, \hat{g}, p \rangle$ .

**Enc** $_{(N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g})}(m)$ : Select  $x \xleftarrow{\$} \mathbb{Z}_N$  and return  $g^m h^x$ .

**Dec** $_{(N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g}, p)}(c)$ : If  $c \in \mathbb{G}$ , then return the discrete logarithm of  $e(c, g)^p$  with respect to base  $\hat{g}$ . If  $c \in \mathbb{G}'$ , then return the discrete logarithm of  $c^p$  with respect to base  $\hat{g}$ .

$c_1 +_{(N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g})} c_2$ : If  $c_1$  and  $c_2$  are in the same group (i.e., both are in  $\mathbb{G}$  or both are in  $\mathbb{G}'$ ), then return  $c_1 c_2$ . Otherwise, if  $c_1 \in \mathbb{G}$  and  $c_2 \in \mathbb{G}'$ , then return  $e(c_1, g) c_2$ .

$m \cdot_{(N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g})} c$ : Return  $c^m$ .

$c_1 \odot_{(N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g})} c_2$ : If  $c_1, c_2 \in \mathbb{G}$ , then return  $e(c_1, c_2)$ . Otherwise, return  $\perp$ .

**Share** $((N, \mathbb{G}, \mathbb{G}', e, g, \hat{g}, p))$ : Return  $sk_1 = \langle \mathbb{G}, \mathbb{G}', d_1 \rangle$  and  $sk_2 = \langle \mathbb{G}, \mathbb{G}', e, g, \hat{g}, d_2 \rangle$  where  $d_1 \xleftarrow{\$} \mathbb{Z}_N$  and  $d_2 \leftarrow p - d_1 \text{ mod } N$ .

**Dec** $^1_{(\mathbb{G}, \mathbb{G}', d_1)}(c)$ : Return  $c^{d_1}$ .

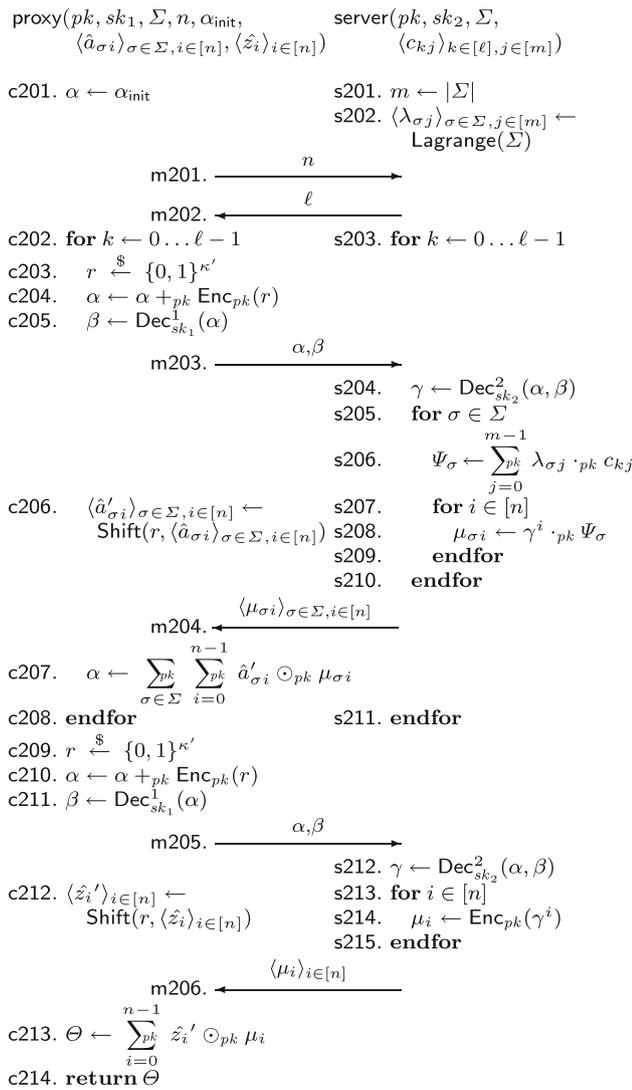
**Dec** $^2_{(\mathbb{G}, \mathbb{G}', e, g, \hat{g}, d_2)}(c_1, c_2)$ : If  $c_1, c_2 \in \mathbb{G}$ , then return the discrete logarithm of  $e(c_2 c_1^{d_2}, g)$  with respect to base  $\hat{g}$ . If  $c_1, c_2 \in \mathbb{G}'$ , then return the discrete logarithm of  $c_2 c_1^{d_2}$  with respect to base  $\hat{g}$ .

Note the new operator  $\odot_{pk}$  that homomorphically multiplies two ciphertexts in  $\mathbb{G}$ . Since the result is in  $\mathbb{G}'$ , it is not possible to use the result as an argument to  $\odot_{pk}$ . This is the sense in which this scheme permits homomorphic multiplication “once”. Also note that though the basic scheme of Boneh et al. did not include  $\hat{g} = e(g, g)^p$  in the public key, Boneh et al. proposed an extension supporting multi-party threshold decryption [36, Section 5] that did so;<sup>3</sup> it is this extension that we adopt here.

A complication of using BGN is the need to compute a discrete logarithm to decrypt in both **Dec** $_{(N, \mathbb{G}, \mathbb{G}', e, g, \hat{g}, p)}$  and **Dec** $^2_{(\mathbb{G}, \mathbb{G}', e, g, \hat{g}, d_2)}$ . We thus need to design our protocol, so that any ciphertext that a party attempts to decrypt should hold a plaintext from a small range  $0 \dots L$ . Then, Pollard’s lambda method [39, p. 128] enables recovery of the plaintext in  $O(\sqrt{L})$  time. Alternatively, a precomputed table that maps  $\hat{g}^m$  to the plaintext  $m \in \{0 \dots L\}$  enables decryption to be performed by table lookup.

**Protocol steps** Protocol  $\Pi_2(\mathcal{E})$  is shown in Fig. 2. It has a similar structure to  $\Pi_1(\mathcal{E})$ , but differs in many respects.

<sup>3</sup> The exact construction supporting threshold decryption was left implicit by Boneh et al. [36], but we have confirmed that including  $\hat{g} = e(g, g)^p$  in the public key is what they intended [38].



**Fig. 2** Protocol  $\Pi_2(\mathcal{E})$ , described in Sect. 3.2

- Rather than taking  $\langle Q, \Sigma, \delta, q_{\text{init}}, \Delta \rangle$  as input, the proxy takes the encrypted initial state  $\alpha_{\text{init}}$  and encrypted coefficients  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$  and  $\langle \hat{z}_i \rangle_{i \in [n]}$  as input. To simplify discussion later, we presume that  $\alpha_{\text{init}}$  is created as a ciphertext of  $q_{\text{init}}$  in  $\mathbb{G}'$ , e.g.,  $\alpha_{\text{init}} \leftarrow \text{Enc}_{pk}(q_{\text{init}}) \odot_{pk} \text{Enc}_{pk}(1)$ . Moreover, Fig. 2 presumes that the coefficients are created by performing  $\langle a_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]} \leftarrow \text{ToPoly}(Q, \Sigma, \delta)$  and  $\langle z_i \rangle_{i \in [n]} \leftarrow \text{ToPoly}(Q, \Sigma, \Delta)$  and then encrypting each coefficient using  $pk$ , i.e.,  $\hat{a}_{\sigma i} \leftarrow \text{Enc}_{pk}(a_{\sigma i})$  for each  $\sigma \in \Sigma$  and  $i \in [n]$ , and  $\hat{z}_i \leftarrow \text{Enc}_{pk}(z_i)$  for each  $i \in [n]$ .
- Because server decrypts the (blinded) DFA state in line s204, the plaintext should be adequately small so that decryption—which as discussed above, involves computing (or looking up) a discrete logarithm if BGN encryption is in use—is not too costly. For this reason, and assuming  $\mathbb{R} = \mathbb{Z}_N$  (as it is in BGN) and  $Q = [n]$ ,

the blinding term  $r$  is drawn from  $\{0, 1\}^{\kappa'}$  instead of  $\mathbb{R}$ , where  $\kappa' \ll \kappa$  is another security parameter. Then, the statistical distance between the distribution of  $\gamma$  seen by server in line s204 when the blinded state is  $q$  (i.e., when  $\gamma = q +_{\mathbb{R}} r$ ) and uniformly random choices from  $\{0, 1\}^{\kappa'}$  is

$$\begin{aligned}
 & \sum_x \left| \mathbb{P}(q + r = x \mid r \xleftarrow{\$} \{0, 1\}^{\kappa'}) - \mathbb{P}(r = x \mid r \xleftarrow{\$} \{0, 1\}^{\kappa'}) \right| \\
 &= \sum_{0 \leq x < q} \frac{1}{2^{\kappa'}} + \sum_{2^{\kappa'} \leq x < q + 2^{\kappa'}} \frac{1}{2^{\kappa'}} = \frac{q}{2^{\kappa'} - 1}
 \end{aligned}$$

Since  $q \in [n]$ , we anticipate setting  $\kappa' = \lceil \log_2 n \rceil + 15$  to achieve a reasonable balance between decryption cost and security for moderately sized  $n$ . In particular, no algorithm can then distinguish the values  $\gamma$  from random elements of  $\{0, 1\}^{\kappa'}$  in a run of the protocol on a file of length  $\ell$  with combined type-I and type-II error better than  $(1 - \frac{1}{2^{15}})^{\ell+1}$  (see [40, Corollary 4]).  $\kappa'$  will thus need to grow with  $n$ , though only logarithmically so. We stress, however, that the need to utilize a  $\kappa'$  at all is a side effect of using BGN encryption in our protocol and is not fundamental to its design.

- The fact that each  $\hat{a}_{\sigma i}$  is a ciphertext necessitates using the “one-time multiplication” operator  $\odot_{pk}$  in line c207 to produce the ciphertext of the new state, versus  $\cdot_{pk}$  as in line c111. The same is true in c213 because each  $\hat{z}_i$  is a ciphertext.
- The protocol returns an encrypted evaluation result  $\Theta$  to the proxy (c214), and so, the original round to decrypt the result (m107–m108) is omitted.

**Protocol security** We are able to prove that  $\Pi_2(\mathcal{E})$  protects DFA privacy and file content privacy against arbitrarily malicious server adversaries, and DFA privacy and file privacy against honest-but-curious proxy adversaries. We do not present the proofs here but refer readers to an accompanying technical report [41] for the proofs. Moreover, since the proxy receives only ciphertexts in the protocol that it cannot decrypt (and the file length  $\ell$ ), we believe it heuristically protects DFA and file privacy against even an arbitrarily malicious proxy. In the next section, we develop an optimized protocol that has better efficiency and achieves similar security properties. We formally define the security notions and evaluate that protocol’s security in Sect. 5.

### 4 Optimizations

In this section, we detail a series of optimizations that we developed for our protocol that, in our implementation, col-

lectively achieved an order-of-magnitude improvement in performance over  $\Pi_2(\mathcal{E})$ .

### 4.1 File representation

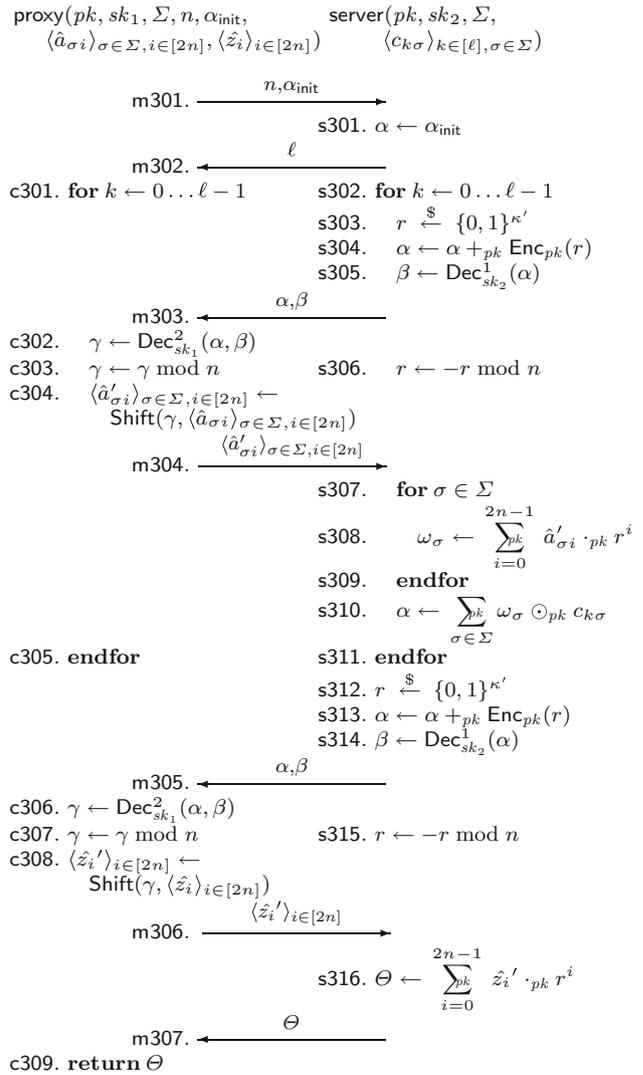
We first observe that, in protocol  $\Pi_2(\mathcal{E})$ , the computation done by the **server** in s206 using the Lagrange coefficients it computed in s202 is effectively evaluating the ciphertext of  $\Lambda_\sigma(\sigma_k)$  for each  $\sigma \in \Sigma$ , using the values  $\langle c_{kj} \rangle_{j \in [m]}$  provided as input to the **server** where  $c_{kj} \leftarrow \text{Enc}_{pk}((\sigma_k)^j)$ . Recall that only for  $\sigma = \sigma_k$  does  $\Lambda_\sigma(\sigma_k) = 1$ ; otherwise,  $\Lambda_\sigma(\sigma_k) = 0$ . Since this calculation only depends on the ciphertexts of the current file character, the result of it, i.e.,  $\langle \text{Enc}_{pk}(\Lambda_\sigma(\sigma_k)) \rangle_{\sigma \in \Sigma}$ , could have been provided by the data owner as *the* ciphertext of file character  $\sigma_k$  so that the **server** would not need to compute it itself.

With this observation, our first optimization is to eliminate the use of the Lagrange polynomial  $\Lambda_\sigma(y)$  completely and decompose the original bivariate polynomial  $f(x, y)$  to  $m$  univariate polynomials, i.e.,  $f_\sigma(x)$  for each  $\sigma \in \Sigma$ . The encryption of a file character  $\sigma_k$  now becomes a vector of encryptions of 0's and one 1. Specifically,  $\sigma_k$  is provided to the **server** as ciphertexts  $\langle c_{k\sigma} \rangle_{\sigma \in \Sigma}$  where  $c_{k\sigma} \leftarrow \text{Enc}_{pk}(1)$  if  $\sigma = \sigma_k$  and  $c_{k\sigma} \leftarrow \text{Enc}_{pk}(0)$  otherwise. This representation has the same storage costs per file character as the original protocol, i.e.,  $m$  ciphertexts per encrypted file character.

### 4.2 Pairing operations

During the implementation of our protocol, we noticed that the pairing operations performed by the **proxy** in c207 are very costly and became the bottleneck for the overall performance. Accelerating pairing operations is a research area of substantial interest, and any progress made would be beneficial to protocols such as ours that utilize pairing. Our focus here, however, is twofold. One is to adapt the protocol to reduce the number of pairing operations. In particular,  $mn$  pairing operations are needed in c207 in each round. In this section, we redesign the protocol to reduce the number of pairing operation down to  $m$ . The other focus is to make the protocol design amenable to pairing preprocessing [42].

Given a bilinear map  $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}'$ , if it is known in advance that a particular value  $c \in \mathbb{G}$  will be paired with other elements multiple times, then preprocessing on  $c$  can be performed in advance to achieve a significant reduction in pairing time. For example, for the class of machines used in our experiments in Sect. 6, a pairing operation for a 1,024-bit BGN scheme costs around 35ms without preprocessing but only 10ms after preprocessing. In c207, the pairing operation performed is  $e(\hat{a}'_{\sigma_i}, \mu_{\sigma_i})$ . Unfortunately, both  $\hat{a}'_{\sigma_i}$  and  $\mu_{\sigma_i}$  change in each round, which prohibits preprocessing. This suggests that performing pairing operations



**Fig. 3** Optimized protocol  $\Pi_3(\mathcal{E})$ , described in Sect. 4

on the **proxy** side may not be the best choice in terms of the potential for optimization.

We therefore redesigned the protocol with the goals of reducing the number of pairing operations and making pairing preprocessing possible. Fortunately, we were able to achieve both goals by shifting the pairing operations to the **server** side. The resulting protocol  $\Pi_3(\mathcal{E})$  is shown in Fig. 3. The new protocol essentially switches the roles of the **proxy** and **server** (though not entirely, since each still receives the same inputs). Note that the directions of the messages m303 and m304 are reversed from those in Fig. 2. The values  $\alpha$  and  $\beta$ , which used to be produced by the **proxy** in c204 and c205, are now produced by the **server** in s304 and s305. This role reversal imposes some significant changes in the computations done by the **proxy** and **server**.

We now describe the changes made in the protocol. We ask the readers to ignore the operations c303, c307, and c306,

§315 for the time being; these will be discussed in Sect. 4.3. The proxy now obtains  $\gamma$  in c302, which is equal to  $q +_{\mathbb{R}} r$ , where  $q$  is the current DFA state and  $r$  was chosen by the server in §303. It now uses  $\gamma$  as input (as opposed to  $r$  in  $\Pi_2(\mathcal{E})$ ) into the Shift procedure first described in Sect. 3.2, i.e., computing new coefficients as:

$$\hat{a}'_{\sigma i} \leftarrow \sum_{i'=0}^{n-1-i} \hat{a}_{\sigma(i+i')} \cdot_{pk} \binom{i+i'}{i'} \cdot_{pk} \gamma^{i'} \tag{6}$$

As a consequence of this “shift”, the plaintexts of the coefficients  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$  define a new polynomial  $f'_\sigma(x)$  such that  $f'_\sigma(x) = f_\sigma(x +_{\mathbb{R}} (q +_{\mathbb{R}} r))$ . The proxy then sends  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$  to the server in m304. The server, knowing  $r$ , blindly “evaluates” the polynomial  $f'_\sigma(x)$  on value  $(-_{\mathbb{R}} r)$  for each  $\sigma \in \Sigma$ , in lines §307–§309. Specifically, it computes a ciphertext of  $f'_\sigma(-_{\mathbb{R}} r) = f_\sigma(-_{\mathbb{R}} r +_{\mathbb{R}} (q +_{\mathbb{R}} r)) = f_\sigma(q)$  as:

$$\omega_\sigma \leftarrow \sum_{i=0}^{n-1} \hat{a}'_{\sigma i} \cdot_{pk} (-_{\mathbb{R}} r)^i \tag{7}$$

A naive way to compute Eq. 7 requires  $O(n^2)$  exponentiations, but by leveraging Horner’s rule, it can be reduced to  $O(n)$  exponentiations. Once the server obtains  $\{\omega_\sigma\}_{\sigma \in \Sigma}$ , it calculates a ciphertext  $\alpha$  of the correct next DFA state in line §310, i.e., by homomorphically summing  $\omega_\sigma \circ_{pk} c_{k\sigma}$  over all  $\sigma \in \Sigma$  (Recall from Sect. 4.1 that, for fixed  $k$ , exactly one of  $\langle c_{k\sigma} \rangle_{\sigma \in \Sigma}$  is a ciphertext of 1 and the rest are ciphertexts of 0). The key point to notice here is that, by rearranging the protocol messages and letting the proxy send over the shifted coefficients, the number of pairing operations is chopped down to only  $m$  from  $nm$ , a major improvement.

We have already alluded to the potential benefit of pairing preprocessing to reduce the online cost of pairing operations. The only question left is how to adapt the protocol, so that it is amenable to using this technique. Fortunately, the changes we have just made to the protocol also makes pairing preprocessing possible. The pairing operation that the server needs to perform in §310 is  $e(\omega_\sigma, c_{k\sigma})$ , for  $\sigma \in \Sigma$  and  $k \in [\ell]$ . The ciphertext  $c_{k\sigma}$  is fixed and known even before the protocol starts. This allows the server to perform pairing preprocessing using these ciphertexts offline and store them to stable storage for future use. During the protocol run, the preprocessing information can be retrieved and used to greatly reduce the online costs of the pairing operations.

### 4.3 Shifting

After the above optimizations, a remaining computation in the protocol that is especially expensive is the Shift procedure, i.e., Eq. 6, which is performed as part of c304 and

c308. Computing each  $\hat{a}'_{\sigma i}$  requires  $O(n)$  exponentiations with exponents being powers of  $\gamma$ . Since  $\gamma$  is  $\kappa'$  bits, this exponentiation is increasingly expensive as  $\kappa'$  grows and is one of the performance bottlenecks of our implementation for the  $\kappa'$  values we employ (As discussed in Sect. 3.2, we take  $\kappa' \approx \lceil \log_2 n \rceil + 15$  in our present implementation, though this setting is an artifact of using BGN encryption and could be larger with another encryption scheme). Our next target is thus to find ways to optimize this operation.

One possibility is to use a smaller  $\kappa'$  to speed up the exponentiations. However,  $\kappa'$  cannot be arbitrarily reduced without reducing the security of the protocol. Instead, here we propose letting the proxy reduce  $\gamma$  modulo  $n$  before feeding it into the Shift procedure, i.e., to compute:

$$\hat{a}'_{\sigma i} \leftarrow \sum_{i'=0}^{n-1-i} \hat{a}_{\sigma(i+i')} \cdot_{pk} \binom{i+i'}{i'} \cdot_{pk} (\gamma \bmod n)^{i'} \tag{8}$$

in Shift, instead (See c303 and c307). By reducing  $\gamma$ , the exponents that used to be  $O(n\kappa')$  bits long are now reduced to  $O(n \log n)$  bits, after taking into account the exponentiations on  $\gamma$  itself. However, this change does have implications for the correctness of the computation. Referring to the derivation in Eq. 7, now that the proxy shifted the polynomial by  $\gamma \bmod n$ , the server needs to adapt to this change accordingly. Intuitively, it should evaluate the new polynomial on  $(-r \bmod n)$  as opposed to on  $-_{\mathbb{R}} r$  in Eq. 7, in which case it computes a ciphertext  $\omega_\sigma$  of

$$\begin{aligned} f'_\sigma(-r \bmod n) &= f_\sigma((-r \bmod n) +_{\mathbb{R}} ((q +_{\mathbb{R}} r) \bmod n)) \\ &= \begin{cases} f_\sigma(q) & \text{if } n \mid r \text{ or } q + (r \bmod n) \geq n \\ f_\sigma(q + n) & \text{otherwise} \end{cases} \end{aligned}$$

assuming that  $\kappa' + 1 < \kappa$  (See lines §306 and §315). However, as indicated, there are two possible outcomes from this calculation. One is exactly what we want, i.e., a ciphertext of  $f_\sigma(q)$ . The other possibility is a ciphertext of  $f_\sigma(q + n)$ , which is problematic because  $f_\sigma(q + n)$  is arbitrary. The server unfortunately cannot tell which case happened because everything it operates on is encrypted. Our solution to this problem is to add constraints when constructing  $f_\sigma(x)$  so that  $f_\sigma(q + n) = f_\sigma(q)$  for all  $q \in Q$  and  $\sigma \in \Sigma$ . These additional constraints guarantee the correct state transition regardless of which case happens. However, the price we pay is that the degree of  $f_\sigma(x)$  increases to  $2n - 1$  since additional  $n$  constraints need to be added to define the polynomial. But the performance gains we achieve outweigh this loss.

Another key insight to draw from this technique is that  $\gamma \bmod n$  can take on only  $n$  different values, and so, there can be at most  $n$  different sets of coefficients  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  from the calculation in Eq. 8. This allows the proxy to precompute

all possible sets of  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  for each  $\gamma \bmod n \in [n]$  and store them in a table before the start of the protocol. It can then simply perform table lookups depending on which value of  $\gamma \bmod n$  it obtains in  $\mathfrak{C303}$  (or  $\mathfrak{C307}$ ). This way, the proxy does not need to perform the computations in Eq. 8 during the protocol except for randomizing the ciphertexts before sending them back to the server. This offers tremendous performance gains for the protocol: Without applying this optimization, the cost for the proxy to calculate Eq. 8 for all  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  involves  $O(mn^2)$  exponentiations in each round. After applying this optimization, it is reduced to only  $O(mn)$  exponentiations, due to the need for ciphertext randomizations.

#### 4.4 Packing the result ciphertexts and boolean combinations

When using  $\Pi_3(\mathcal{E})$  to evaluate a DFA on  $k$  files,  $k$  encrypted evaluation results  $\Theta_0, \dots, \Theta_{k-1}$ —each the ciphertext of a 0 or 1—need to be communicated back to the user. Sending these  $k$  ciphertexts individually to the user introduces an undesirably high communication cost between the proxy and the user. A better approach is for the proxy to aggregate multiple such results into a single ciphertext before sending these results back to the user. Specifically, the proxy can aggregate these  $k$  ciphertexts into ciphertexts  $\Theta_0, \dots, \Theta_{\lceil k/z \rceil - 1}$  where

$$\Theta_i \leftarrow \sum_{j=0}^{z-1} 2^j \cdot_{pk} \Theta_{iz+j}$$

for each  $i \in [\lceil k/z \rceil]$ . This aggregation is omitted from Fig. 3. The user can then decrypt each  $\Theta_i$  to recover all the evaluation results. The value of  $z$  is upper bounded by the bit length of the plaintext space of the cryptosystem  $\mathcal{E}$  in general, and in the case of BGN,  $z$  must be restricted to a small value such as  $\kappa'$  to enable efficient decryption by the user.

This packing technique generalizes nicely to support evaluating conjunctions or disjunctions of  $d$  DFAs on  $k$  files. That is, after the proxy interacts with the server to evaluate DFAs  $M_0, \dots, M_d$  on each of  $k$  files, yielding encrypted results  $\Theta_{0,0}, \dots, \Theta_{k-1,d-1}$ , the proxy can aggregate these  $kd$  ciphertexts into ciphertexts  $\Theta_0, \dots, \Theta_{\lceil k/z' \rceil - 1}$  where  $z' = \lfloor z / \lceil \log_2(d+1) \rceil \rfloor$  and

$$\Theta_i \leftarrow \sum_{j=0}^{z'-1} \left( 2^{j \lceil \log_2(d+1) \rceil} \cdot_{pk} \sum_{d'=0}^{d-1} \Theta_{iz'+j,d'} \right)$$

for each  $i \in [\lceil k/z' \rceil]$ . Upon decrypting each such aggregate ciphertext, each  $\lceil \log_2(d+1) \rceil$ -length sequence of bits represents the number of DFAs  $M_0, \dots, M_{d-1}$  that the corresponding file matched. That is, the file satisfies the disjunction of these DFAs if that count is nonzero, and it satisfies the conjunction of these DFAs if that count is  $d$ .

To evaluate other Boolean combinations of  $d$  DFAs on files, it suffices for the proxy and server to evaluate each DFA individually on each file and communicate the results per DFA to the user, and the user can herself determine which files match the Boolean combination she is interested in. While less communication efficient than the above approach for conjunctions and disjunctions, this approach is more computationally efficient for the proxy and server than combining all  $d$  DFAs into a single large DFA that represents the Boolean combination of interest.

### 5 Protocol security

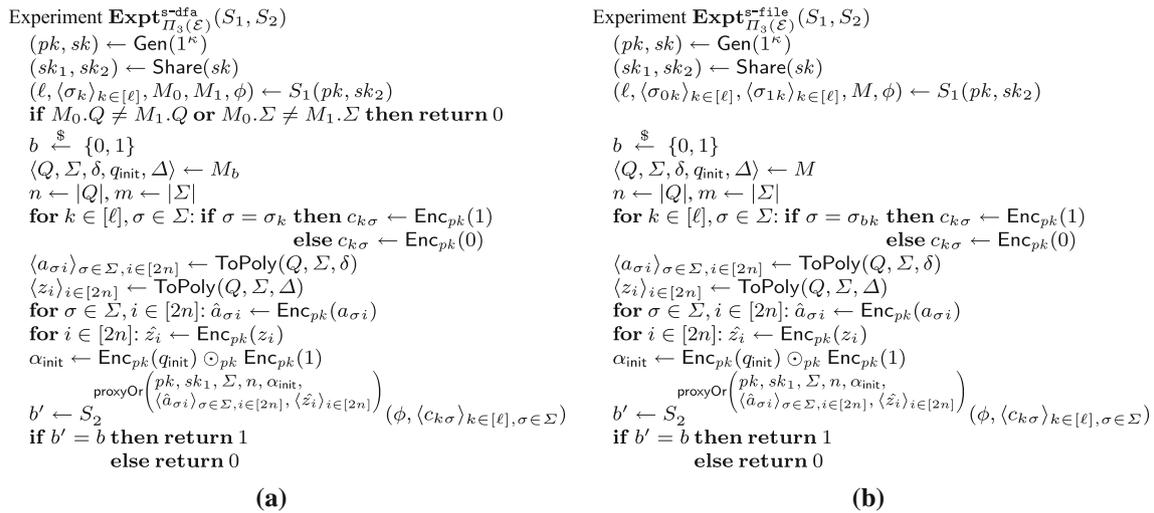
In this section, we analyze the security of  $\Pi_3(\mathcal{E})$  as shown in Fig. 3 (For simplicity, here, we elide consideration of the extension in Sect. 4.4, not shown in Fig. 3, though it has no impact on the security of the protocol). We show that the protocol  $\Pi_3(\mathcal{E})$  provably protects the privacy of both the DFA and file contents from either honest-but-curious proxy adversaries or arbitrarily malicious server adversaries.

#### 5.1 Security against server adversaries

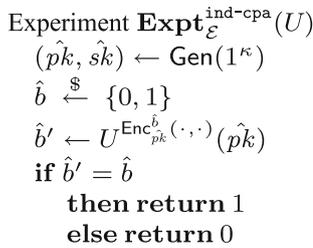
In this section, we bound the advantage that an arbitrarily malicious server gains by executing this protocol, in determining either the DFA that the proxy is evaluating or the plaintext of the file in its possession. That is, we prove the privacy of the file and DFA inputs against server adversaries.

Following previous security definitions [15], we formalize our security claims against server compromise by defining two separate server adversaries. The first server adversary  $S = (S_1, S_2)$  attacks the encrypted DFA  $M = \langle Q, \Sigma, \delta, q_{\text{init}}, \Delta \rangle$ , i.e.,  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [n]}$  held by the proxy, as described in experiment  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-dfa}}$  in Fig. 4a.  $S_1$  first generates a file  $\langle \sigma_k \rangle_{k \in [\ell]}$  and two DFAs  $M_0, M_1$ . (Note that we use, e.g., “ $M_0.Q$ ” and “ $M_1.Q$ ” to disambiguate their state sets.)  $S_2$  is then invoked with the ciphertexts  $\langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma}$  of its file and information  $\phi$  created for it by  $S_1$  and is given oracle access to proxyOr. proxyOr is given input arguments pertaining to one of the two DFAs output by  $S_1$ , selected at random (as indicated by  $b$ ).

proxyOr responds to queries from  $S_2$  as follows, ignoring malformed queries. The first query (say, consisting of simply “start”) causes proxyOr to begin the protocol; proxyOr responds with a message of the form  $n, \alpha_{\text{init}}$  (i.e., of the form of message m301). The second invocation by  $S_2$  must include a single integer  $\ell$  (i.e., of the form of message m302). The next  $\ell$  queries by  $S_2$  must be of the form  $\alpha, \beta$ , i.e., two values as in message m303, to which proxyOr responds by sending  $2nm$  elements of  $C_{pk}$ , i.e.,  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  as in m304.  $S_2$ 's next query to proxyOr again must contain two values of the form  $\alpha, \beta$  (as in m305), to which proxyOr responds with  $2n$



**Fig. 4** Experiments for proving security of  $\Pi_3(\mathcal{E})$  against server adversaries. **a** Experiment  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-dfa}}$ . **b** Experiment  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-file}}$



**Fig. 5**  $\text{Expt}_{\mathcal{E}}^{\text{ind-cpa}}(U)$

ciphertexts  $\langle \hat{z}_i \rangle_{i \in [2n]}$  as in m306. The next (and last) query by  $S_2$  can consist one element of  $C_{pk}$  as in m307. Eventually,  $S_2$  outputs a bit  $b'$ , and  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-dfa}}(S) = 1$  only if  $b' = b$ . We say the *advantage* of an arbitrarily malicious  $S$  is

$$\text{Adv}_{\Pi_3(\mathcal{E})}^{\text{s-dfa}}(S) = 2 \cdot \mathbb{P} \left( \text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-dfa}}(S) = 1 \right) - 1$$

and define  $\text{Adv}_{\Pi_3(\mathcal{E})}^{\text{s-dfa}}(t, \ell, n, m) = \max_S \text{Adv}_{\Pi_3(\mathcal{E})}^{\text{s-dfa}}(S)$  where the maximum is taken over all adversaries  $S$  taking time  $t$  and selecting a file of length  $\ell$  and DFAs containing  $n$  states and an alphabet of  $m$  symbols.

We reduce DFA privacy against server attacks to the IND-CPA [43] security of the encryption scheme. IND-CPA security is defined using the experiment in Fig. 5, in which an adversary  $U$  is provided a public key  $\hat{pk}$  and access to an oracle  $\text{Enc}_{\hat{pk}}^{\hat{b}}(\cdot, \cdot)$  that consistently encrypts either the first of its two inputs (if  $\hat{b} = 0$ ) or the second of those inputs (if  $\hat{b} = 1$ ). Eventually,  $U$  outputs a guess  $\hat{b}'$  at  $\hat{b}$ , and  $\text{Expt}_{\mathcal{E}}^{\text{ind-cpa}}(U) = 1$  only if  $\hat{b}' = \hat{b}$ . The IND-CPA advantage of  $U$  is defined as

$$\text{Adv}_{\mathcal{E}}^{\text{ind-cpa}}(U) = 2 \cdot \mathbb{P} \left( \text{Expt}_{\mathcal{E}}^{\text{ind-cpa}}(U) = 1 \right) - 1$$

and then  $\text{Adv}_{\mathcal{E}}^{\text{ind-cpa}}(t, w) = \max_U \text{Adv}_{\mathcal{E}}^{\text{ind-cpa}}(U)$  where the maximum is taken over all adversaries  $U$  executing in time  $t$  and making  $w$  queries to  $\text{Enc}_{\hat{pk}}^{\hat{b}}(\cdot, \cdot)$ .

In our theorem statements, we omit terms that are negligible as a function of the security parameters  $\kappa$  and  $\kappa'$ . For any  $\mathcal{E}$  operation  $\text{op}$ , we use  $t_{\text{op}}$  to denote the time required to perform  $\text{op}$ ; e.g.,  $t_{\text{Dec}}$  is the time to perform a Dec operation.

**Theorem 1** For  $t' = t + t_{\text{Share}} + (\ell m + 2nm + 2n + 1) \cdot t_{\text{Enc}}$ ,

$$\text{Adv}_{\Pi_3(\text{BGN})}^{\text{s-dfa}}(t, \ell, n, m) \leq n^{\ell+1} \text{Adv}_{\text{BGN}}^{\text{ind-cpa}}(t', 2nm + 2n + 1)$$

*Proof* Given an adversary  $S = (S_1, S_2)$  for  $\Pi_3(\text{BGN})$  that runs in time  $t$ , produces a file of length  $\ell$ , and produces DFAs of  $n$  states over an alphabet of  $m$  symbols, we construct an IND-CPA attacker  $U$  for BGN to demonstrate the theorem as follows. On input a BGN public key  $\hat{pk} = \langle N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g} \rangle$ ,  $U$  sets  $d_2 \stackrel{\$}{\leftarrow} \mathbb{Z}_N$ , and invokes  $S_1(\hat{pk}, sk_2)$  where  $sk_2 = \langle \mathbb{G}, \mathbb{G}', d_2 \rangle$  to obtain  $(\ell, \langle \sigma_k \rangle_{k \in [\ell]}, M_0, M_1, \phi)$ . Note that  $d_2$  is chosen from a distribution that is perfectly indistinguishable from that from which  $d_2$  is chosen in the real system. If  $M_0.Q \neq M_1.Q$  or  $M_0.\Sigma \neq M_1.\Sigma$ , then  $U$  aborts the simulation. Otherwise, letting  $\Sigma = M_0.\Sigma$ ,  $m = |\Sigma|$  and  $n = |M_0.Q|$ ,  $U$  computes  $\langle a_{0\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]} \leftarrow \text{ToPoly}(M_0.Q, \Sigma, M_0.\delta)$  and  $\langle a_{1\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]} \leftarrow \text{ToPoly}(M_1.Q, \Sigma, M_1.\delta)$ , and it sets  $\hat{a}_{\sigma i} \leftarrow \text{Enc}_{\hat{pk}}^{\hat{b}}(a_{0\sigma i}, a_{1\sigma i})$  for  $\sigma \in \Sigma$  and  $i \in [n]$ . It then computes  $\langle z_{0i} \rangle_{i \in [2n]} \leftarrow \text{ToPoly}(M_0.Q, \Sigma, M_0.\Delta)$  and  $\langle z_{1i} \rangle_{i \in [2n]} \leftarrow \text{ToPoly}(M_1.Q, \Sigma, M_1.\Delta)$ , and it sets  $\hat{z}_i \leftarrow \text{Enc}_{\hat{pk}}^{\hat{b}}(z_{0i}, z_{1i})$  for  $i \in [n]$ .  $U$  finally sets  $\alpha_{\text{init}} \leftarrow \text{Enc}_{\hat{pk}}^{\hat{b}}(M_0.q_{\text{init}}, M_1.q_{\text{init}}) \odot_{\hat{pk}} \text{Enc}_{\hat{pk}}^{\hat{b}}(1)$ , and then for all  $k \in [\ell], \sigma \in \Sigma$ , it sets  $c_{k\sigma} \leftarrow \text{Enc}_{pk}(1)$  if  $\sigma = \sigma_k$  and  $c_{k\sigma} \leftarrow \text{Enc}_{pk}(0)$  otherwise.

$U$  then invokes  $S_2(\phi, \langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma})$  and simulates responses to  $S_2$ 's queries to **proxyOr** as follows (ignoring malformed invocations). Upon initializing  $S_2$ ,  $U$  sends  $n, \alpha_{\text{init}}$  to  $S_2$  and gets  $\ell$  in return. For the  $k$ th query of the form  $\alpha, \beta$  ( $0 \leq k < \ell$ ),  $U$  selects  $\gamma \xleftarrow{\$} [n]$ , as opposed to decrypting it as in a real execution (see c302 and c303). It computes  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]} \leftarrow \text{Shift}(\gamma, \langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]})$  as in c304 and returns  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  to  $S_2$ . For the  $(\ell + 1)$ th query of the form  $\alpha, \beta$ ,  $U$  again randomly sets  $\gamma \xleftarrow{\$} [n]$  and  $\langle \hat{z}'_i \rangle_{i \in [2n]} \leftarrow \text{Shift}(\gamma, \langle \hat{z}_i \rangle_{i \in [2n]})$  and then sends  $\langle \hat{z}'_i \rangle_{i \in [2n]}$  to  $S_2$ . Finally, when  $S_2$  outputs  $b'$ ,  $U$  outputs  $b'$ , as well.

$U$ 's simulation is perfectly indistinguishable from the real system to a malicious server adversary  $S$  if and only if  $U$  made the correct guesses on  $\gamma$  in each round. When that happens, the advantage of  $U$  winning his game is the same with that of  $S$ . So,  $\text{Adv}_{\text{BGN}}^{\text{ind-cpa}}(U) \geq (\frac{1}{n})^{\ell+1} \text{Adv}_{\Pi_3(\text{BGN})}^{\text{s-dfa}}(S)$ .  $U$  runs in time  $t' = t + t_{\text{Share}} + \ell m \cdot t_{\text{Enc}} + (2nm + 2n + 1) \cdot t_{\text{Enc}}$  due to the need to generate a secret key share for  $S$ , to generate  $\langle c_{kj} \rangle_{k \in [\ell], j \in [m]}$ , and to make  $2nm + 2n + 1$  encryption oracle queries to create  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$ ,  $\langle \hat{z}_i \rangle_{i \in [2n]}$  and  $\alpha_{\text{init}}$ .  $\square$

The multiplicative factor of  $n^{\ell+1}$  that appears in Theorem 1 and in Theorem 2 below, while independent of the security parameters  $\kappa$  and  $\kappa'$ , nevertheless, renders these theorems of limited practical use. That said, we have no reason to believe that the actual security of  $\Pi_3(\text{BGN})$  against server adversaries decays so dramatically as a function of  $\ell$ . Rather, this factor is simply an artifact of our proof method, and since the server in  $\Pi_3(\text{BGN})$  receives (aside from the value  $n$ ) only ciphertexts from the proxy created using a public key for which it does not hold the private key, we believe these theorems to be overwhelmingly conservative.

In addition, the  $n^{\ell+1}$  factor can be eliminated in both theorems if  $S$  is honest-but-curious and so follows the protocol as prescribed, provided that we modify the protocol with the following trick: the data owner initializes the server with another, distinct public key—for which the corresponding private key is given to nobody—that the server uses to encrypt each value of  $r$  it selects in the protocol (s303 and s312), sending this ciphertext along with  $\alpha, \beta$  to the proxy in messages m303 and m305. This ciphertext is useless to the proxy since it does not hold the key to decrypt it. However, in the proof of security against an honest-but-curious server, who faithfully includes  $r$  in this ciphertext, the IND-CPA adversary  $U$  can create this new public key for the server adversary  $S$ , while keeping the private key for decrypting the ciphertexts containing the  $r$  values. This enables completion of a security proof for an honest-but-curious  $S$  that does not incur a  $n^{\ell+1}$  loss factor. Indeed, a similar trick was used in our proof of security for protocol  $\Pi_1(\mathcal{E})$  [15].

The second server adversary  $S = (S_1, S_2)$  that we consider attacks the file for which it holds the per-symbol ciphertexts  $\langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma}$  as in experiment  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-file}}$  shown in

Fig. 4b. Here,  $S_1$  produces two separate, equal-length plaintext files  $\langle \sigma_{0k} \rangle_{k \in [\ell]}$ ,  $\langle \sigma_{1k} \rangle_{k \in [\ell]}$  and a DFA  $M$ .  $S_2$  then receives ciphertexts  $\langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma}$  for file  $\langle \sigma_{bk} \rangle_{k \in [\ell]}$  where  $b$  is chosen randomly.  $S_2$  is also given oracle access to **proxyOr**( $pk, sk_1, \Sigma, n, \alpha_{\text{init}}, \langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}, \langle \hat{z}_i \rangle_{i \in [2n]}$ ). The interaction between  $S_2$  and **proxyOr** is similar to what was described for the server DFA adversary. Eventually,  $S_2$  outputs a bit  $b'$ , and  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-file}}(S) = 1$  iff  $b' = b$ . The advantage of  $S$  is

$$\text{Adv}_{\Pi_3(\mathcal{E})}^{\text{s-file}}(S) = 2 \cdot \mathbb{P}(\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{s-file}}(S) = 1) - 1$$

and then  $\text{Adv}_{\Pi_3(\mathcal{E})}^{\text{s-file}}(t, \ell, n, m) = \max_S \text{Adv}_{\Pi_3(\mathcal{E})}^{\text{s-file}}(S)$  where the maximum is taken over all adversaries  $S = (S_1, S_2)$  taking time  $t$  and producing (from  $S_1$ ) files of  $\ell$  symbols and a DFA of  $n$  states and alphabet of size  $m$ .

**Theorem 2** For  $t' = t + t_{\text{Share}} + (\ell m + 2nm + 2n + 1) \cdot t_{\text{Enc}}$ ,

$$\text{Adv}_{\Pi_3(\text{BGN})}^{\text{s-file}}(t, \ell, n, m) \leq n^{\ell+1} \text{Adv}_{\text{BGN}}^{\text{ind-cpa}}(t', \ell m)$$

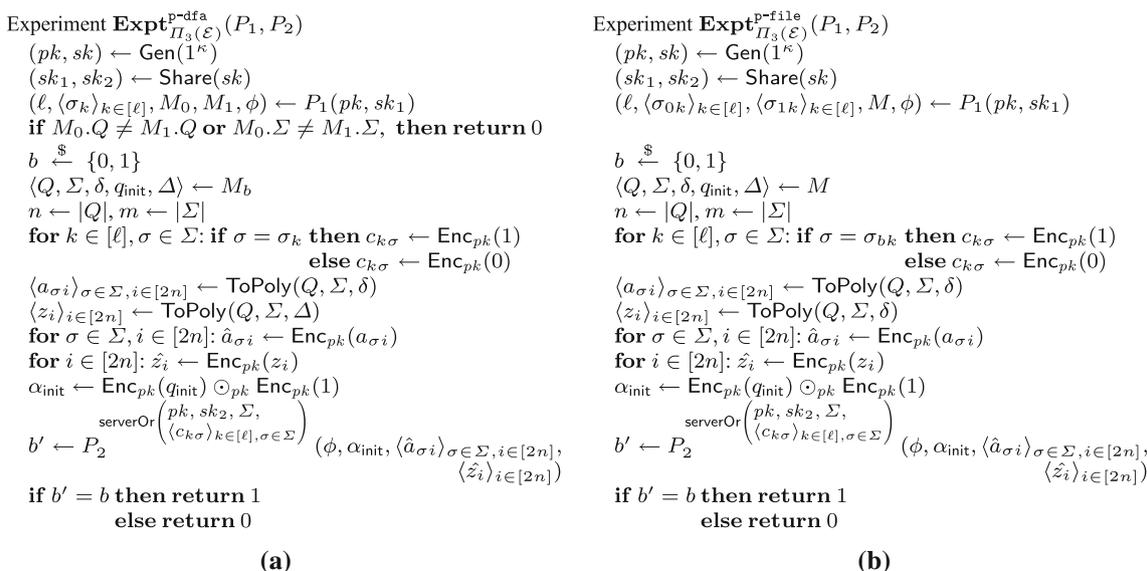
*Proof* Given an adversary  $S = (S_1, S_2)$  running in time  $t$  and selecting files of length  $\ell$  symbols and a DFA of  $n$  states over an alphabet of  $m$  symbols, we construct an IND-CPA adversary  $U$ . On input a BGN public key  $\hat{pk} = \langle N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g} \rangle$ ,  $U$  sets  $d_2 \xleftarrow{\$} \mathbb{Z}_N$ , and invokes  $S_1(\hat{pk}, sk_2)$  where  $sk_2 = \langle \mathbb{G}, \mathbb{G}', d_2 \rangle$  to obtain  $(\ell, \langle \sigma_{0k} \rangle_{k \in [\ell]}, \langle \sigma_{1k} \rangle_{k \in [\ell]}, M, \phi)$ , where  $M = \langle Q, \Sigma, q_{\text{init}}, \delta, \Delta \rangle$  is a DFA. Note that  $d_2$  is chosen from a distribution that is perfectly indistinguishable from that from which  $d_2$  is chosen in the real system. For  $k \in [\ell]$  and  $\sigma \in \Sigma$ ,  $U$  sets  $c_{kj} \leftarrow \text{Enc}_{\hat{pk}}^{\hat{c}}(I_0, I_1)$  where

$$I_0 \leftarrow \begin{cases} 1 & \text{if } \sigma = \sigma_{0k} \\ 0 & \text{otherwise} \end{cases} \quad I_1 \leftarrow \begin{cases} 1 & \text{if } \sigma = \sigma_{1k} \\ 0 & \text{otherwise} \end{cases}$$

$U$  also sets  $\alpha_{\text{init}} \leftarrow \text{Enc}_{\hat{pk}}^{\hat{c}}(q_{\text{init}}) \odot_{\hat{pk}} \text{Enc}_{\hat{pk}}^{\hat{c}}(1, \langle a_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}) \leftarrow \text{ToPoly}(Q, \Sigma, \delta)$ , and  $\langle z_i \rangle_{i \in [2n]} \leftarrow \text{ToPoly}(Q, \Sigma, \Delta)$ .  $U$  then computes  $\hat{a}_{\sigma i} \leftarrow \text{Enc}_{\hat{pk}}^{\hat{c}}(a_{\sigma i})$  and  $\hat{z}_i \leftarrow \text{Enc}_{\hat{pk}}^{\hat{c}}(z_i)$  for all  $\sigma \in \Sigma$  and  $i \in [2n]$ .

$U$  then invokes  $S_2(\phi, \langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma})$  and simulates responses to  $S_2$ 's queries to **proxyOr** as follows (ignoring malformed invocations). Upon initializing  $S_2$ ,  $U$  sends  $n, \alpha_{\text{init}}$  to  $S_2$  and gets  $\ell$  in return. For the  $k$ th query of the form  $\alpha, \beta$  ( $0 \leq k < \ell$ ),  $U$  selects  $\gamma \xleftarrow{\$} [n]$ , as opposed to decrypting it as in a real execution c302 and c303.  $U$  then sets  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]} \leftarrow \text{Shift}(\gamma, \langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]})$  as done in c304 and returns  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  to  $S_2$ . For the  $(\ell + 1)$ th query of the form  $\alpha, \beta$ ,  $U$  again randomly sets  $\gamma \xleftarrow{\$} [n]$  and then computes  $\langle \hat{z}'_i \rangle_{i \in [2n]} \leftarrow \text{Shift}(\gamma, \langle \hat{z}_i \rangle_{i \in [2n]})$  and sends  $\langle \hat{z}'_i \rangle_{i \in [2n]}$  to  $S_2$ . Finally, when  $S_2$  outputs  $b'$ ,  $U$  outputs  $b'$ .

This simulation is perfectly indistinguishable from the real system provided that  $U$  made correct guesses for  $\gamma$  on each round of the simulation. When that happens,  $U$  wins his game if and only if  $S$  wins his. So, we have  $\text{Adv}_{\text{BGN}}^{\text{ind-cpa}}(U) \geq (\frac{1}{n})^{\ell+1} \text{Adv}_{\Pi_3(\text{BGN})}^{\text{s-file}}(S)$ .  $U$  runs in time  $t' = t + t_{\text{Share}} +$



**Fig. 6** Experiments for proving security of  $\Pi_3(\mathcal{E})$  against proxy adversaries. **a** Experiment  $\mathbf{Expt}_{\Pi_3(\mathcal{E})}^{\text{p-dfa}}$ . **b** Experiment  $\mathbf{Expt}_{\Pi_3(\mathcal{E})}^{\text{p-file}}$

$(2nm + 2n + 1) \cdot t_{\text{Enc}} + \ell m \cdot t_{\text{Enc}}$  due to the need to generate a secret key share for  $S$ , to generate  $\alpha_{\text{init}}$ ,  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  and  $\langle \hat{z}_i \rangle_{i \in [2n]}$ , and to make  $\ell m$  queries to its encryption oracle to create  $\langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma}$ .  $\square$

### 5.2 Security against proxy adversaries

In this section, we analyze the privacy of the DFA and file from proxy adversaries, specifically *honest-but-curious* ones. Our protocol’s security is limited to honest-but-curious proxies as an artifact of using BGN encryption, specifically because this forces us to employ  $\kappa' \ll \kappa$ . Advances in additively homomorphic encryption that also supports “one-time” homomorphic multiplication and that also permits us to employ  $\kappa' \approx \kappa$ , would permit us to prove security against malicious proxy adversaries, as well.<sup>4</sup>

The case of proxy adversaries in  $\Pi_3(\mathcal{E})$  differs more substantially from that in prior work [15]. For one, we need to formalize and prove a result about the degree to which the DFA is protected from the proxy. Such an experiment for defining this type of security is shown in Fig. 6a. In this experiment,  $P_2$  is invoked with encrypted coefficients  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$ ,  $\langle \hat{z}_i \rangle_{i \in [2n]}$  and the encrypted initial state  $\alpha_{\text{init}}$  for one of two DFAs output by  $P_1$  (determined by random selection of  $b$ ).  $P_2$  can invoke `serverOr` first with an integer  $n$  and a ciphertext (as in m301), in response to which `serverOr` returns  $\ell$  (as in m302). In the next  $\ell$  rounds, each time `serverOr` sends ciphertext  $\alpha$  and partial decryption  $\beta$  (as in m303) to  $P_2$ .  $P_2$  then sends ciphertexts  $\langle \hat{a}'_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$  in response (as

in m304). The next round, `serverOr` again sends  $\alpha$  and  $\beta$  (as in m305) to  $S_2$ , who responds with ciphertexts  $\langle \hat{z}'_i \rangle_{i \in [2n]}$  as in m306. `serverOr` sends one last message consisting one element in  $C_{pk}$  to  $S_2$  as in m307. Finally,  $P_2$  outputs a bit  $b'$ , and  $\mathbf{Expt}_{\Pi_3(\mathcal{E})}^{\text{p-dfa}}(P) = 1$  only if  $b' = b$ .

We prove DFA privacy against *honest-but-curious proxy* adversaries. A proxy adversary  $(P_1, P_2)$  is honest-but-curious if  $P_2$  invokes `serverOr` exactly as  $\Pi_3(\mathcal{E})$  prescribes. The honest-but-curious advantage  $\mathbf{hbcAdv}_{\Pi_3(\mathcal{E})}^{\text{p-dfa}}(P)$  of adversary  $P = (P_1, P_2)$  is

$$\mathbf{hbcAdv}_{\Pi_3(\mathcal{E})}^{\text{p-dfa}}(P) = 2 \cdot \mathbb{P}(\mathbf{Expt}_{\Pi_3(\mathcal{E})}^{\text{p-file}}(P) = 1) - 1$$

and  $\mathbf{hbcAdv}_{\Pi_3(\mathcal{E})}^{\text{p-dfa}}(t, \ell, n, m) = \max_P \mathbf{Adv}_{\Pi_3(\mathcal{E})}^{\text{p-dfa}}(P)$  where the maximum is taken over all honest-but-curious client adversaries  $P$  running in total time  $t$  and producing files of length  $\ell$  and a DFA of  $n$  over an alphabet of  $m$  symbols.

We now discuss the DFA privacy against an honest-but-curious proxy adversary.

**Theorem 3** For  $t' = t + t_{\text{Share}} + (\ell m + 2nm + 2n + 2) \cdot t_{\text{Enc}}$ ,  $\mathbf{hbcAdv}_{\Pi_3(\text{BGN})}^{\text{p-dfa}}(t, \ell, n, m) \leq \mathbf{Adv}_{\text{BGN}}^{\text{ind-cpa}}(t', 2(nm + n + 1))$

*Proof* Given an adversary  $P = (P_1, P_2)$  running in time  $t$  and selecting files of length  $\ell$  and a DFA of  $n$  states over an alphabet of  $m$  symbols, we construct an IND-CPA adversary  $U$ . On input a BGN public key  $\hat{pk} = \langle N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g} \rangle$ ,  $U$  sets  $d_1 \stackrel{\$}{\leftarrow} \mathbb{Z}_N$ , and invokes  $P_1(\hat{pk}, sk_1)$  where  $sk_1 = \langle \mathbb{G}, \mathbb{G}', d_1 \rangle$  to obtain  $(\ell, \langle \sigma_k \rangle_{k \in [\ell]}, M_0, M_1, \phi)$ . Note that  $d_1$  is chosen from a distribution that is perfectly indistinguishable from that from which  $d_1$  is chosen in the real system. If  $M_0.Q \neq M_1.Q$  or  $M_0.\Sigma \neq M_1.\Sigma$ , then  $U$  aborts the simulation. Letting  $\Sigma = M_0.\Sigma$ ,  $m = |\Sigma|$  and  $n = |M_0.Q|$ ,

<sup>4</sup> For example, if  $r \stackrel{\$}{\leftarrow} \mathbb{R}$  in lines s303 and s312, then  $\gamma$  in lines c302 and c306 is uniformly distributed in  $\mathbb{R}$  and so conveys no information to the proxy, regardless of how the proxy misbehaves.

$U$  computes  $\langle a_{0\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]} \leftarrow \text{ToPoly}(M_0.Q, \Sigma, M_0.\delta)$  and  $\langle a_{1\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]} \leftarrow \text{ToPoly}(M_1.Q, \Sigma, M_1.\delta)$ , and then sets  $\hat{a}_{\sigma i} \leftarrow \text{Enc}_{\hat{p}k}^b(a_{0\sigma i}, a_{1\sigma i})$  for  $\sigma \in \Sigma$  and  $i \in [2n]$ . It also computes  $\langle z_{0i} \rangle_{i \in [2n]} \leftarrow \text{ToPoly}(M_0.Q, \Sigma, M_0.\Delta)$  and  $\langle z_{1i} \rangle_{i \in [2n]} \leftarrow \text{ToPoly}(M_1.Q, \Sigma, M_1.\Delta)$ , and then sets  $\hat{z}_i \leftarrow \text{Enc}_{\hat{p}k}^b(z_{0i}, z_{1i})$  for  $i \in [2n]$ .  $U$  then sets  $\alpha_{\text{init}} \leftarrow \text{Enc}_{\hat{p}k}^b(M_0.q_{\text{init}}, M_1.q_{\text{init}}) \odot_{\hat{p}k} \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(1)$ . For all  $k \in [\ell]$ ,  $\sigma \in \Sigma$ ,  $U$  also sets  $c_{k\sigma} \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(1)$  if  $\sigma = \sigma_k$  and  $c_{k\sigma} \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(0)$  otherwise.

$U$  invokes  $P_2(\phi, \alpha_{\text{init}}, \langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}, \langle \hat{z}_i \rangle_{i \in [2n]})$  and simulates responses to  $P_2$ 's queries to `serverOr` as follows. Upon receiving the first message from  $P_2$ ,  $U$  sends back  $\ell$ . In each round,  $U$  sets  $r \xleftarrow{\$} \{0, 1\}^{\kappa'}$ ,  $\alpha \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(r) \odot_{\hat{p}k} \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(1)$  and  $\beta \leftarrow \hat{g}^r \alpha^{-d_1}$  so that  $\alpha^{d_1} \beta = \hat{g}^r$ . It then sends  $\alpha$  and  $\beta$  to  $P_2$ . In the last round, upon receiving  $\langle \hat{z}_i \rangle_{i \in [n]}$  as in m306,  $U$  sets  $\Theta \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(M_0(\langle \sigma_k \rangle_{k \in [\ell]}), M_1(\langle \sigma_k \rangle_{k \in [\ell]}))$  where  $M_0(\langle \sigma_k \rangle_{k \in [\ell]})$  denotes the evaluation result of  $M_0$  on the file and similarly for  $M_1(\langle \sigma_k \rangle_{k \in [\ell]})$ .  $U$  then sends  $\Theta$  to  $P_2$ . Finally, when  $P_2$  outputs  $b'$ ,  $U$  outputs  $b'$  as well.

$U$ 's simulation is statistically indistinguishable (as a function of  $\kappa'$ ) from a real protocol execution as long as  $P_2$  is honest-but-curious. So  $\text{Adv}_{\text{BGN}}^{\text{ind-cpa}}(U) \geq \text{hbcAdv}_{\Pi_3(\text{BGN})}^{\text{p-dfa}}(P)$ .  $U$  runs in time  $t' = t + t_{\text{Share}} + \ell m \cdot t_{\text{Enc}} + (2nm + 2n + 2) \cdot t_{\text{Enc}}$  due to the need to generate a secret key share for  $P$ , to create  $\langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma}$ , and to make  $2nm + 2n + 2$  queries to its encryption oracle in order to generate  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$ ,  $\langle \hat{z}_i \rangle_{i \in [2n]}$ ,  $\alpha_{\text{init}}$  and  $\Theta$  in the final round.  $\square$

Next, we consider security against attacks on the encrypted files from a proxy adversary. Since the proxy no longer learns the final state of the DFA evaluation, we do not require the proxy adversary to choose two files that produce the same final result for the user's DFA, compared to previous definitions [15]. The experiment that we use to define file security against proxy adversaries  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{p-file}}$  is shown in Fig. 6b. There,  $P_1$  produces two separate, equal-length plaintext files  $\langle \sigma_{0k} \rangle_{k \in [\ell]}$ ,  $\langle \sigma_{1k} \rangle_{k \in [\ell]}$  and a DFA  $M$ .  $P_2$  then receives the ciphertexts  $\langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma}$  for file  $\langle \sigma_{bk} \rangle_{k \in [\ell]}$  where  $b$  is chosen randomly.  $P_2$  is also given oracle access to `serverOr` and finally  $P_2$  outputs a bit  $b'$ , and  $\text{Expt}_{\Pi_3(\mathcal{E})}^{\text{p-file}}(P) = 1$  iff  $b' = b$ . The advantage of an honest-but-curious adversary  $P = (P_1, P_2)$  is defined as:

$$\text{hbcAdv}_{\Pi_3(\mathcal{E})}^{\text{p-file}}(P) = 2 \cdot \mathbb{P} \left( \text{Expt}_{\Pi_3(\mathcal{E})}^{\text{p-file}}(P) = 1 \right) - 1$$

and  $\text{hbcAdv}_{\Pi_3(\mathcal{E})}^{\text{p-file}}(t, \ell, n, m) = \max_P \text{hbcAdv}_{\Pi_3(\mathcal{E})}^{\text{p-file}}(P)$  where the maximum is taken over all honest-but-curious proxy adversaries  $P$  running in total time  $t$  and producing files of length  $\ell$  and a DFA of  $n$  over an alphabet of  $m$  symbols. We now have:

**Theorem 4** For  $t' = t + t_{\text{Share}} + (2nm + 2n + \ell m + 2) \cdot t_{\text{Enc}}$ ,

$$\text{hbcAdv}_{\Pi_3(\text{BGN})}^{\text{p-file}}(t, \ell, n, m) \leq \text{Adv}_{\text{BGN}}^{\text{ind-cpa}}(t', \ell m + 1)$$

*Proof* Given an adversary  $P = (P_1, P_2)$  running in time  $t$  and selecting files of length  $\ell$  and a DFA of  $n$  states over an alphabet of  $m$  symbols, we construct an IND-CPA adversary  $U$ . On input a BGN public key  $\hat{p}k = \langle N, \mathbb{G}, \mathbb{G}', e, g, h, \hat{g} \rangle$ ,  $U$  sets  $d_1 \xleftarrow{\$} \mathbb{Z}_N$ , and invokes  $P_1(\hat{p}k, sk_1)$  where  $sk_1 = \langle \mathbb{G}, \mathbb{G}', d_1 \rangle$  to obtain  $(\ell, \langle \sigma_{0k} \rangle_{k \in [\ell]}, \langle \sigma_{1k} \rangle_{k \in [\ell]}, M, \phi)$ . Note that  $d_1$  is chosen from a distribution that is perfectly indistinguishable from that from which  $d_1$  is chosen in the real system. Let  $\Sigma = M.\Sigma$ ,  $Q = M.Q$ ,  $\Delta = M.\Delta$ ,  $m = |\Sigma|$  and  $n = |Q|$ . For  $k \in [\ell]$  and  $\sigma \in \Sigma$ ,  $U$  sets  $c_{kj} \leftarrow \text{Enc}_{\hat{p}k}^b(I_0, I_1)$  where

$$I_0 \leftarrow \begin{cases} 1 & \text{if } \sigma = \sigma_{0k} \\ 0 & \text{otherwise} \end{cases} \quad I_1 \leftarrow \begin{cases} 1 & \text{if } \sigma = \sigma_{1k} \\ 0 & \text{otherwise} \end{cases}$$

$U$  also sets  $\alpha_{\text{init}} \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(q_{\text{init}}) \odot_{\hat{p}k} \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(1)$ ,  $\langle a_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]} \leftarrow \text{ToPoly}(Q, \Sigma, \delta)$ , and  $\langle z_i \rangle_{i \in [2n]} \leftarrow \text{ToPoly}(Q, \Sigma, \Delta)$ .  $U$  then computes  $\hat{a}_{\sigma i} \leftarrow \text{Enc}_{\hat{p}k}^b(a_{\sigma i})$  and  $\hat{z}_i \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(z_i)$  for all  $\sigma \in \Sigma$  and  $i \in [2n]$ .

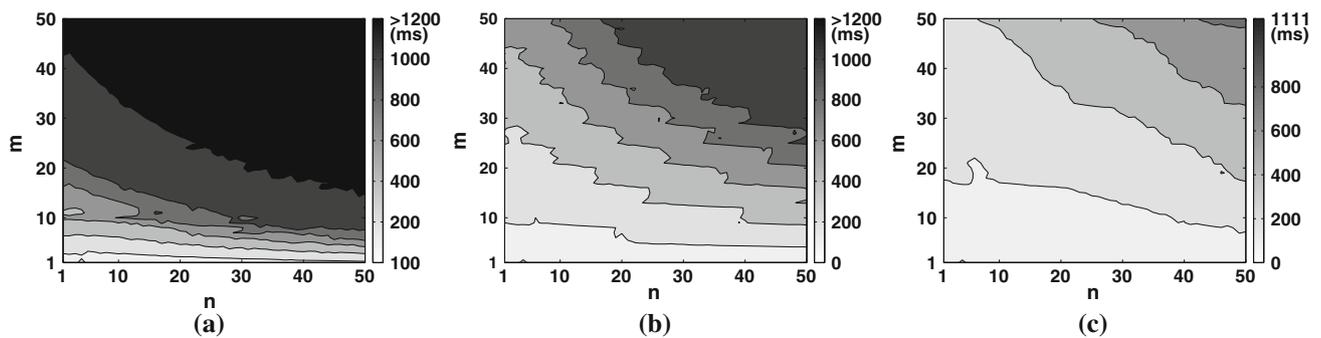
$U$  invokes  $P_2(\phi, \alpha_{\text{init}}, \langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}, \langle \hat{z}_i \rangle_{i \in [2n]})$  and simulates responses to  $P_2$ 's queries to `serverOr` as follows. Upon receiving the first message from  $P_2$ ,  $U$  sends back  $\ell$ . In each round,  $U$  sets  $r \xleftarrow{\$} \{0, 1\}^{\kappa'}$ ,  $\alpha \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(r) \odot_{\hat{p}k} \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(1)$  and  $\beta \leftarrow \hat{g}^r \alpha^{-d_1}$  so that  $\alpha^{d_1} \beta = \hat{g}^r$ . It then sends  $\alpha$  and  $\beta$  to  $P_2$ . In the last round, upon receiving  $\langle \hat{z}_i \rangle_{i \in [n]}$  as in m306,  $U$  sets  $\Theta \leftarrow \text{Enc}_{\hat{p}k}^{\hat{\cdot}}(M(\langle \sigma_{0k} \rangle_{k \in [\ell]}), M(\langle \sigma_{1k} \rangle_{k \in [\ell]}))$ .  $U$  then sends  $\Theta$  to  $P_2$ . Finally, when  $P_2$  outputs  $b'$ ,  $U$  outputs  $b'$ .

$U$ 's simulation is statistically indistinguishable (as a function of  $\kappa'$ ) from a real protocol execution as long as  $P_2$  is honest-but-curious. So  $\text{Adv}_{\text{BGN}}^{\text{ind-cpa}}(U) \geq \text{hbcAdv}_{\Pi_3(\text{BGN})}^{\text{p-file}}(P)$ .  $U$  runs in time  $t' = t + t_{\text{Share}} + (2nm + 2n + 1) \cdot t_{\text{Enc}} + (\ell m + 1) \cdot t_{\text{Enc}}$  due to the need to generate a secret key share for  $P$ , to generate  $\langle \hat{a}_{\sigma i} \rangle_{\sigma \in \Sigma, i \in [2n]}$ ,  $\langle \hat{z}_i \rangle_{i \in [2n]}$  and  $\alpha_{\text{init}}$ , and to make  $\ell m + 1$  queries to its encryption oracle in order to create  $\langle c_{k\sigma} \rangle_{k \in [\ell], \sigma \in \Sigma}$  and  $\Theta$  in the last round.  $\square$

## 6 Performance evaluation

### 6.1 Implementation

We implemented our optimized protocol  $\Pi_3(\mathcal{E})$  in Java using an open source Java pairing-based cryptography library jPBC [44], which is built on the original C pairing library PBC [42]. Our regular-expression-to-DFA conversion engine is built around the Java dk.brics.automaton library [45]. The complete implementation contains about 5,000 physical source lines of code.



**Fig. 7** Time spent per file character in milliseconds, with pairing preprocessing disabled. **a** 1 thread per worker. **b** 4 threads per worker. **c** 16 threads per worker

For our evaluation, we chose a BGN public key of size  $\kappa = 1,024$  and a secondary security parameter of  $\kappa' = 22$ . To further improve performance, we utilized a fixed-base windowing exponentiation technique [39] to accelerate exponentiation operations on the proxy side. We also take advantage of pairing preprocessing at the server (see Sect. 4.2) and compare the performance with and without this optimization. In particular, pairing preprocessing produces approximately 600 KB of information per BGN ciphertext and so increases the required storage dramatically. As such, it may not be appropriate for use in some environments.

To exploit parallelism available in the protocol computation and the physical hardware, we implemented two levels of parallelization for the server and proxy programs. The first level is a thread pool of workers, each running a single server or proxy instance. Each server worker grabs an encrypted email from a shared queue of all the encrypted emails being searched and runs a protocol instance with its paired proxy worker independently. Each server or proxy worker can further spawn extra threads to assist in its computation. This level of parallelization is designed to take advantage of the computational independence found in many calculations in the protocol. For example, for server workers, the calculation in line s308 can be split among multiple threads before combining the results to obtain  $\alpha$ . Similarly for client workers, the Shift procedure in line c304 can also be partitioned across multiple threads.

## 6.2 Microbenchmarks

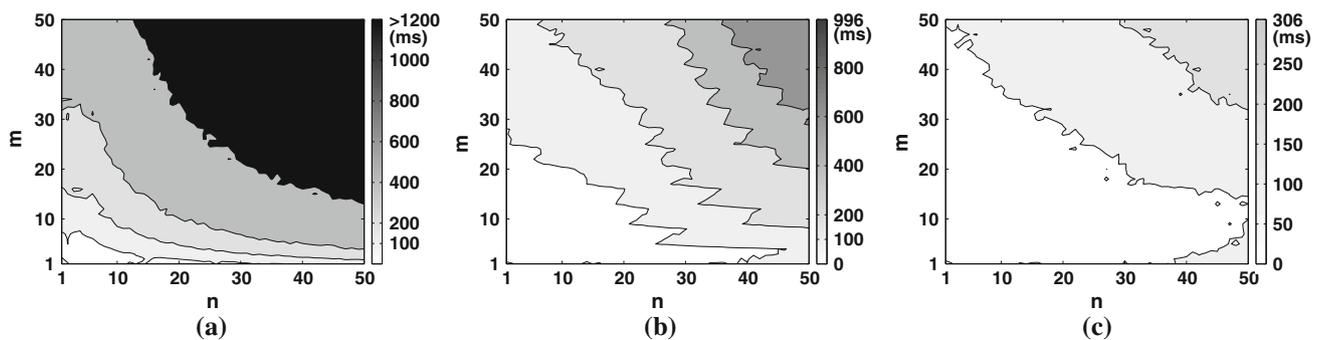
We first report microbenchmarks for our implementation. The experiments reported below were conducted using two machines connected via a 1 Gb/s network, each equipped with two quad-core Intel Xeon 2.67 GHz CPUs with simultaneous multithreading enabled. All proxy workers ran on one of these machines, and all server workers ran on the other.

To understand the performance cost of the protocol and the impact of its two parameters, i.e., the number of DFA

states  $n$  and the alphabet size  $m$ , we conducted experiments measuring the average time spent by the server and proxy for processing one character (or one round of protocol execution) for various combinations of  $n$  and  $m$ . For this purpose, we generated encrypted files each consisting of 20 characters for  $m = 1$  to  $m = 50$ . We then created random DFAs with number of states  $n$  ranging from 1 to 50 and ran them against the files. We computed the average time spent per character by dividing the total time spent processing a file by 20. The results, with pairing preprocessing disabled, are presented in contour graphs in Fig. 7 where the times are binned into ranges, each shown as a band representing the range indicated in the sidebar legend.

To show that the computation of the protocol is highly parallelizable, in this experiment, we launched a single worker on both server and proxy machines and tested its performance when 1, 4, and 16 threads are spawned by each worker to assist in their computations, shown in Fig. 7a–c, respectively. It is clear from all three graphs that the protocol performance scales much better with the increase of  $n$  than with  $m$ ; the number of expensive pairing operations performed by the server in each round is equal to  $m$ , and the cost resulted from the increase of  $m$  significantly outweighs that resulting from the increase of  $n$ . These results also show that the protocol is highly amenable to parallelization, with dramatically decreased processing time as the number of threads increases.

Since Fig. 7 clearly shows the impact of the pairing operations on the overall performance of the protocol, we went on to evaluate how much improvement pairing preprocessing can provide. In these experiments, we applied preprocessing on the file ciphertexts before conducting the same experiments as described above. The results are shown in Fig. 8. As expected, the overall protocol performance improves significantly in each of the multithreading cases, with darker bands reduced dramatically in size. More importantly, the protocol performance now scales much better with the increase of  $m$  because of the significantly reduced cost of pairing operations on the server side.



**Fig. 8** Time spent per file character in milliseconds, with pairing preprocessing enabled. **a** 1 thread per worker. **b** 4 threads per worker. **c** 16 threads per worker

To better understand the relative computational burden imposed on the **server** and **proxy** by the protocol, we also measured the average CPU time spent processing one character for the **server** and **proxy** processes for each combination of  $n$  and  $m$ . To perform these tests, we instantiated one **server** worker and one **proxy** worker, each with a single thread. The CPU time takes into account the amount of time spent in both user and kernel modes. The results for the **server** and **proxy** are plotted in Fig. 9a, c, respectively. For the **server** side, it generally takes below or around 100ms for one round of computation when  $m$  is less than 10. The **proxy** side enjoys a slightly lighter computational cost and spends around or below 100ms even for  $m$  as high as 50 with  $n$  less than 15. The results reveal that the **server** side takes more hit when  $m$  increases due to the need to perform the pairing operations, while the **proxy** achieves a more balanced degradation with the increasing of  $n$  or  $m$ .

We also conducted the same experiments when pairing preprocessing is used on the file ciphertexts in advance. Since it does not affect the **proxy** side processing time, we only show the **server** side CPU time in Fig. 9b. Compared with Fig. 9a, the CPU time spent is reduced significantly and the performance scales better as  $m$  increases.

Since the protocol is interactive, we also measured the aggregate network bandwidth consumption between the **server** and **proxy** in one round of protocol execution. As shown in Fig. 9d, the bandwidth usage ranges from about 15 KB per round (i.e., per file character) for moderate  $n$  and  $m$  to as much as 640 KB per round when  $n$  and  $m$  are 50.

### 6.3 Case study: regular-expression search on encrypted emails

To further provide insight into the expected performance when using our protocol in real-world applications, we conducted a case study for performing regular-expression search on public key encrypted emails. We envision an email system in which the sender encrypts the email body using a traditional hybrid encryption scheme, in which the email body

is encrypted using a symmetric encryption key that itself is encrypted by the receiver's public key. To enable search operations, however, the sender also attaches an encrypted searchable "header" to the encrypted email body that consists of all the information from the email that allows searching. We now detail the design of this header.

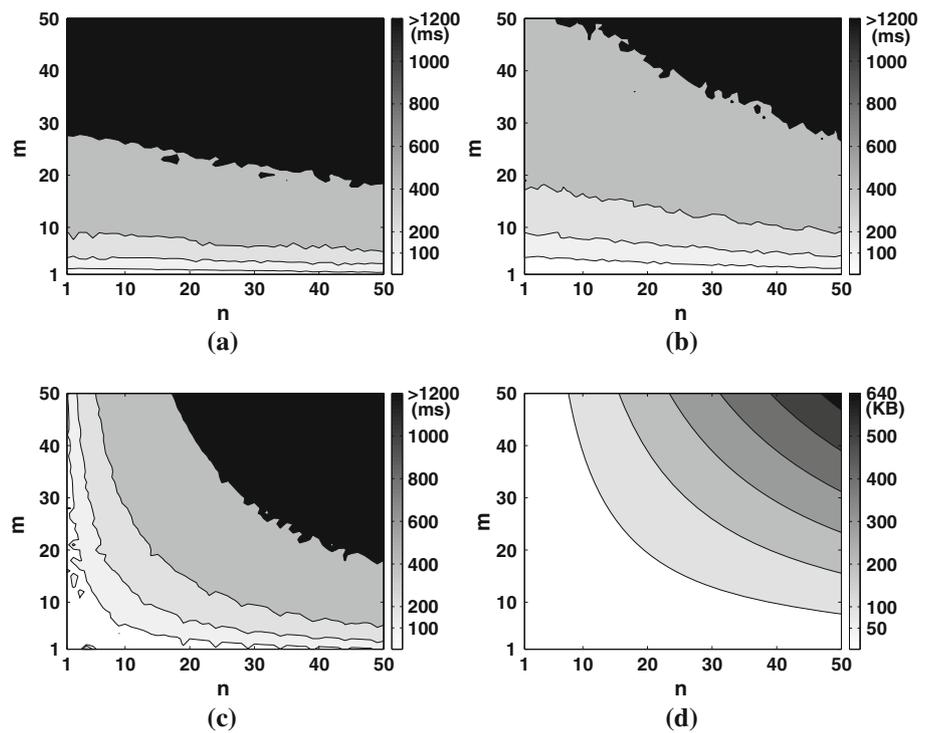
#### 6.3.1 Header information

Our current design allows searching on selected header fields of the email that are most commonly searched: (1) date; (2) sender email address; (3) sender name; and (4) subject line. The character-by-character encryption of the four headers is attached to the encrypted email body to enable searches. Since characters in each header are usually drawn from different distributions, we define the *dictionary* of a header as the set containing all possible characters and field-specific words that can be used in that header. Each header-field text is encoded using the dictionary before encryption, including sanitizing any characters not present in the dictionary (e.g., converting uppercase letters to lowercase, if only lowercase are included in the dictionary). We stress that this sanitization is only applied to the encrypted header that facilitates the search operations. The original field value in the email body is left intact.

The benefit of defining different dictionaries for different header fields is that adding field-specific words provides an opportunity for compressing the header fields, which is reminiscent of dictionary-based compression schemes. In addition, we envision dictionaries to be receiver-specific, e.g., distributed within the public key certificate for the receiver. Below, we describe how each dictionary is defined for each header in our evaluation.

**Date:** Date is converted into YYMMDD, where year, month and day each consists of two digits of numerical values. For years, we expect to store emails dated from 1990 to 2050. So, we included "90" to "50", as words, in the dictionary to encode the year. Similarly for months and days, we also

**Fig. 9** CPU time and network bandwidth measurements. **a** server CPU time per file character, with pairing preprocessing disabled. **b** server CPU time per file character, with pairing preprocessing enabled. **c** proxy CPU time per file character. **d** Network bandwidth per file character



added “01” to “31” into the dictionary. So, the dictionary is defined as  $\{00, 01, 02, \dots, 49, 50\} \cup \{90, 91, \dots, 99\}$ .

**Sender address:** The sender email address is represented in the usual format, e.g., “alice@abc.com”, where the dictionary consists of “a” through “z”, “0” through “9”, “.”, “@”, “\_”, and “-”. We also added into the dictionary several common email service names like “gmail”, domain names like “com”, and “enron” because the email dataset used in our evaluation (see Sect. 6.3.3) was from Enron. (Again, dictionaries can be receiver-specific). In total, the dictionary for this field is  $\{a, \dots, z\} \cup \{0, \dots, 9\} \cup \{., @, _, -\} \cup \{gmail, yahoo, aol, hotmail, enron, com, edu, net, org\}$ .

**Sender name:** The sender name field represents the sender’s name with first name followed by the last name, separated by a space. The name dictionary consists of “a” through “z” and the space character.

**Subject line:** The subject line is allowed to include arbitrary characters that can be typed from a keyboard. However, in practice, users rarely create search queries including special characters [1]. So, in our design, we restrict the dictionary to include “a” through “z”, “0” through “9”, and selected special characters including “@”, “!”, “%”, “.”, and the space character.

### 6.3.2 Encoding

To enable DFA evaluations, we need to define the input alphabet  $\Sigma$  that drives the DFA state transitions. The simplest way

is to define it as the union of all the dictionaries defined for all header fields, which would result in an  $m$  well above 50. However, the experiment results in Sect. 6.2 suggested that the protocol performance is very sensitive to a large  $m$ . So, we make the DFA alphabet  $\Sigma$  and its size  $m$  a user-defined parameter and designed a method to encode each word in the dictionary into a representation using the input symbols in  $\Sigma$ . Since the exact representation of the input symbols in  $\Sigma$  is not important, for simplicity, we use numerical values to represent each symbol. For example, a size  $m$  alphabet will consists of  $\Sigma = \{0, 1, \dots, m - 1\}$ . Then, each word in a dictionary is represented using a distinct sequence of symbols from  $\Sigma$ . Each of the header fields is first encoded using this method and then encrypted symbol by symbol. The regular-expression query is encoded in the same way before converting it into a DFA.

### 6.3.3 Evaluation

In order to shed light on the expected performance when using our protocol to perform search operations in real-world email systems, we implemented a prototype search system and evaluated its performance based on the Enron email dataset [46], which is a publicly available real-world email corpus that contains roughly 0.6 million messages from about 150 then-employees of Enron. We randomly sampled 1,000 emails from the inboxes of all the users in the dataset and performed evaluations using selected representative search

queries. In the experiments, we fixed a DFA alphabet of size  $m = 4$ .

Motivated by the email search features found in ThunderBird [47], we selected four different queries to evaluate the protocol efficiency. For the date field, we selected a range query to search for all emails with date stamps between 2001/09/10 and 2002/04/20. The corresponding regular expression is

```
(0109(10|11|...|31)) | (01(10|11|12)(01|...|31))
| (02|(01|02|03)(01|...|31)) | (0204(01|02|...|20))
```

which results in a DFA of  $n = 23$  states using our conversion engine. For the sender address field, we selected a query to search for emails with sender address ending with the string “enron.com”. The resulting regular expression is `*enron.com` where `*` denotes zero or more occurrences of dictionary words, which converts into a DFA with  $n = 9$  states. For the name field, we selected the query to search for sender name containing the word “John”, which translates into the regular expression `*John*`, with a corresponding DFA containing  $n = 17$  states. Lastly for the subject line field, we chose to search for emails with subject lines containing the word “meet” followed by “Jan” followed by a space and two arbitrary characters. This translates into a regular expression of `*meet Jan ??*` where `?` denotes exactly one occurrence of a dictionary word, which results in a DFA of  $n = 36$  states.

We encrypted the bodies of 1,000 randomly selected emails using GnuPG [48], which results in an average size of 1.5 KB per email. We wrote our own tool to generate the encrypted searchable headers, which take up about 185 KB per email. To understand the performance impact when using the two parallelization techniques described in Sect. 6.1, we report performance numbers for various combinations of the number of workers and the number of threads that each worker spawns. The average time spent processing each email is shown in Table 1, which was calculated by dividing the total time to finish processing all 1,000 emails by 1,000. In order to demonstrate the performance improvement when pairing preprocessing is applied on email ciphertexts, we also performed pairing preprocessing for the email ciphertexts and stored the results on disk. The resulting protocol performance measurements (after preprocessing) are shown in the same table inside parentheses. The performance gain is very compelling, as it offers an approximately 30% improvement over the version without preprocessing. However, the downside is that it needs significantly more storage space to store the pairing preprocessing information.

The experiment results also demonstrate the benefit of concurrently processing multiple emails by instantiating multiple workers. In most cases, doubling the number of workers results in a decrease of the timing results by a factor of two. Meanwhile, spawning multiple threads for each worker has similar effect, although to a lesser extent. This can

**Table 1** Average time per email in seconds; numbers in parenthesis are when pairing preprocessing was applied on email ciphertexts in advance

Workers	Threads per worker		
	1	2	4
(a) Query ‘*enron.com’ on sender address field			
1	13.09 (6.95)	8.00 (4.69)	6.75 (5.26)
2	6.68 (3.58)	3.97 (2.72)	3.45 (2.68)
4	3.31 (1.81)	2.01 (1.51)	2.01 (1.44)
8	1.70 (0.98)	1.40 (0.93)	1.34 ( <b>0.89</b> )
16	1.27 (0.94)	<b>1.25</b> (0.94)	1.26 (0.95)
(b) Range query 2001/09/10–2002/04/20 on date field			
1	3.55 (2.38)	2.03 (1.39)	1.23 (0.99)
2	1.68 (1.17)	0.96 (0.71)	0.63 (0.50)
4	0.84 (0.57)	0.49 (0.36)	0.42 (0.28)
8	0.43 (0.30)	0.30 (0.20)	0.29 (0.19)
16	0.27 (0.18)	0.26 (0.18)	<b>0.25 (0.17)</b>
(c) Query ‘*John*’ on sender name field			
1	12.34 (7.93)	7.55 (4.84)	4.84 (3.50)
2	6.47 (3.99)	3.56 (2.40)	2.40 (1.96)
4	3.13 (1.98)	1.82 (1.28)	1.56 (1.16)
8	1.62 (1.05)	1.25 (0.80)	1.15 (0.77)
16	1.09 ( <b>0.79</b> )	<b>1.06</b> (0.81)	1.06 (0.81)
(d) Query ‘*meet Jan ??*’ on subject field			
1	35.97 (27.17)	20.17 (15.02)	11.38 (9.52)
2	17.68 (12.85)	9.49 (7.41)	5.75 (4.80)
4	8.59 (6.30)	4.81 (3.71)	4.20 (2.97)
8	4.41 (3.21)	2.81 (2.10)	3.04 (1.91)
16	2.49 (1.93)	2.55 ( <b>1.75</b> )	<b>2.39</b> (1.77)

Best results are indicated in bold

be seen by reading the entries horizontally, where the timing results are typically reduced by about 40% as the number of threads per worker doubles. The date query (Table 1b) finishes fastest, averaging (in the best worker/thread configuration) only a quarter of a second (s) to process one email and 0.17s when pairing preprocessing is used. This is due to the fact that the date field is very short for all emails. The sender name query (Table 1c) averaged 1.06s per email and 0.79s with pairing preprocessing. This is followed by the sender address query (Table 1a), which averaged 1.25s per email and 0.89s with pairing preprocessing. The subject line query (Table 1d) was the slowest, mainly due to the fact that subject lines in the email corpus are usually much longer than the other fields.

## 7 Conclusion

In this paper, we developed a novel protocol to perform private regular-expression evaluations on encrypted data stored

at a **server**. The protocol allows a **user** to delegate a regular expression search request to a **proxy**, which interacts with the **server** and returns the search result to the **user**. The privacy of the search query is provably protected against both an honest-but-curious **proxy** and an arbitrarily malicious **server** (provided that they do not maliciously collude), and so, is the privacy of the data itself. In Sect. 4, we described a number of optimization steps for the protocol that resulted in a substantial performance improvement for it, while retaining the protocol's security properties. Many of these techniques should apply to the previous protocol [15] (summarized in Sect. 3.1) that provided our starting point for this work. We demonstrated the performance of our protocol using a working implementation, in both microbenchmarks and when applied to a real-world dataset (Sect. 6).

**Acknowledgments** This work was supported in part by grants 0831245 and 0910483 from the National Science Foundation.

**Open Access** This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

## References

- Harvey, M., Elswiler, D.: Exploring query patterns in email search. In: 34th European Conference on Advances in Information Retrieval, 2012, pp. 25–36
- Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
- Goyvaerts, J., Levithan, S.: Regular Expressions Cookbook, 2nd edn. O'Reilly Media Inc., Sebastopol (2012)
- Rivest, R., Adleman, L., Dertouzos, M.: On Data banks and Privacy Homomorphisms. Foundations of Secure Computation, pp. 169–177. Academic Press, Inc (1978)
- Yao, A.C.: Protocols for secure computations. In: 23rd IEEE Symposium on Foundations of Computer Science, pp. 160–164 (1982)
- Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: 19th ACM Symposium on Theory of Computing, pp. 218–229 (1987)
- Gentry, C.: Fully homomorphic encryption using ideal lattices. In: 41st ACM Symposium on Theory of Computing, pp. 169–178 (2009)
- van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. Adv. Cryptol. EUROCRYPT 2010, 24–43 (2010)
- Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public Key Cryptography—PKC 2010, May 2010, pp. 420–443
- Stehlé, D., Steinfeld, R.: Faster fully homomorphic encryption. In: Advances in Cryptology—ASIACRYPT. Dec 2010, pp. 377–394 (2010)
- Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay—a secure two-party computation system. In: 13th USENIX Security Symposium, Aug 2004, pp. 287–302
- Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: 15th ACM Conference on Computer and Communications Security, pp. 257–266 (2008)
- Pinkas, B., Schneider, T., Smart, N., Williams, S.: Secure two-party computation is practical. In: Advances in Cryptology—ASIACRYPT. Dec 2009, pp. 250–267 (2009)
- Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: 20th USENIX Security Symposium, pp. 35–35 (2011)
- Wei, L., Reiter, M.K.: Third-party private DFA evaluation on encrypted files in the cloud. In: Computer Security—ESORICS 2012: 17th European Symposium on Research in Computer Security (2012)
- Blanton, M., Aliasgari, M.: Secure outsourcing of DNA searching via finite automata. In: Data and Applications Security and Privacy XXIV, June 2010, pp. 49–64
- Song, D., Wagner, D., Perrig, A.: Practical techniques for searching on encrypted data. In: 2000 IEEE Symposium on Security and Privacy, May 2000, pp. 44–55
- Goh, E.-J.: Secure indexes. Cryptology ePrint Archive, Report 2003/216 (2003). <http://eprint.iacr.org/>
- Boneh, D., Crescenzo, G.D., Ostrovsky, R., Persiano, G.: Public key encryption with keyword search. In: Advances in Cryptology—EUROCRYPT 2004, ser. Lecture Notes in Computer Science, vol. 3027, pp. 506–522 (2004)
- Chang, Y.-C., Mitzenmacher, M.: Privacy preserving keyword searches on remote encrypted data. In: Applied Cryptography and Network Security, 3rd International Conference, pp. 442–455 (2005)
- Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: 13th ACM Conference on Computer and Communications Security, pp. 79–88 (2006)
- Boneh, D., Waters, B.: Conjunctive, subset, and range queries on encrypted data. In: 4th Theory of Cryptography Conference, Feb 2007, pp. 535–554
- Katz, J., Sahai, A., Waters, B.: Predicate encryption supporting disjunctions, polynomial equations, and inner products. In: Advances in Cryptology—EUROCRYPT, Apr 2008, pp. 146–162 (2008)
- Kamara, S., Papamanthou, C., Roeder, T.: Dynamic searchable symmetric encryption. In: 19th ACM Conference on Computer and Communications Security, Oct 2012, pp. 965–976
- Abdalla, M., Bellare, M., Catalano, D., Kiltz, E., Kohno, T., Lange, T., Malone-Lee, J., Neven, G., Paillier, P., Shi, H.: Searchable encryption revisited: consistency properties, relation to anonymous IBE, and extensions. J. Cryptol. 21(3), 350–391 (2008)
- Baek, J., Safavi-Naini, R., Susilo, W.: Public key encryption with keyword search revisited. In: Computational Science and Its Applications—ICCSA 2008, ser. Lecture Notes in Computer Science, vol. 5072, pp. 1249–1259 (2008)
- Rhee, H.S., Park, J.H., Susilo, W., Lee, D.H.: Improved searchable public key encryption with designated tester. In: 4th ACM Conference on Information, Computer, and Communications Security, March 2009, pp. 376–379
- Troncoso-Pastoriza, J.R., Katzenbeisser, S., Celik, M.: Privacy preserving error resilient DNA searching through oblivious automata. In: 14th ACM Conference on Computer and Communications Security, pp. 519–528 (2007)
- Frikken, K.B.: Practical private DNA string searching and matching through efficient oblivious automata evaluation. In: Data and Applications Security XXIII, July 2009, pp. 81–94
- Gennaro, R., Hazay, C., Sorensen, J.S.: Text search protocols with simulation based security. In: Public Key Cryptography—PKC 2010, pp. 332–350 (2010)
- Mohassel, P., Niksefat, S., Sadeghian, S., Sadeghiyan, B.: An efficient protocol for oblivious DFA evaluation and applications. In: Topics in Cryptology—CT-RSA 2012, pp. 398–415 (2012)

32. Choi, S.G., Elbaz, A., Juels, A., Malkin, T., Yung, M.: Two-party computing with encrypted data. In: *Advances in Cryptology—ASIACRYPT 2007*, pp. 298–314 (2007)
33. Waters, B.: Functional encryption for regular languages. In: *Advances in Cryptology—CRYPTO 2012*, pp. 218–235 (2012)
34. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: *Advances in Cryptology—EUROCRYPT '99*, May 1999, pp. 223–238
35. Damgård, I., Jurik, M.: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In: *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptosystems*, Feb 2001, pp. 119–136
36. Boneh, D., Goh, E.-J., Nissim, K.: Evaluating 2-DNF formulas on ciphertexts. In: *2nd Theory of Cryptography Conference*, pp. 325–342 (2005)
37. Gentry, C., Halevi, S., Vaikuntanathan, V.: A simple BGN-type cryptosystem from LWE. In: *Advances in Cryptology—EUROCRYPT 2010*, May 2010, pp. 506–522
38. Boneh, D.: Personal communication, July 2011
39. Menezes, A.J., Oorschot, P.C.V., Vanstone, S.A.: *Handbook of Applied Cryptography*. CRC Press, Boca Raton (1997)
40. Baignères, T., Vaudenay, S.: The complexity of distinguishing distributions (invited talk). In: *Information Theoretic Security, 3rd International Conference*, pp. 210–222 (2008)
41. Wei, L., Reiter, M.K.: Third-party DFA evaluation on encrypted files. Department of Computer Science, UNC Chapel Hill, Technical report TR11-005 (2011)
42. Lynn, B.: The pairing based cryptography library (2006). <http://crypto.stanford.edu/pbc/>
43. Bellare, M., Desai, A., Pointcheval, D., Rogaway, P.: Relations among notions of security for public-key encryption schemes. In: *Advances in Cryptology—CRYPTO '98*, Aug 1998, pp. 26–45
44. Caro, A.D.: Java pairing based cryptography library (2012). <http://gas.dia.unisa.it/projects/jpbc/>
45. Moller, A.: dk.brics.automaton (2011). <http://www.brics.dk/automaton/>
46. Enron email dataset (2009). <http://www.cs.cmu.edu/~enron/>
47. Mozilla Thunderbird (2005). <https://www.mozilla.org/en-US/thunderbird/>
48. The GNU privacy guard (1998). <http://www.gnupg.org/>