Ensuring File Authenticity in Private DFA Evaluation on Encrypted Files in the Cloud

Lei Wei and Michael K. Reiter

Department of Computer Science
University of North Carolina at Chapel Hill
{lwei,reiter}@cs.unc.edu

Abstract. Cloud storage, and more specifically the encryption of file contents to protect them in the cloud, can interfere with access to these files by partially trusted third-party service providers and customers. To support such access for pattern-matching applications (e.g., malware scanning), we present a protocol that enables a client authorized by the data owner to evaluate a deterministic finite automaton (DFA) on a file stored at a server (the cloud), even though the file is encrypted by the data owner for protection from the server. Our protocol contributes over previous work by enabling the client to detect any misbehavior of the server; in particular, the client can verify that the result of its DFA evaluation is based on the file stored there by the data owner, and in this sense the file and protocol result are authenticated to the client. Our protocol also protects the privacy of the file and the DFA from the server, and the privacy of the file (except the result of evaluating the DFA on it) from the client. A special case of our protocol solves private DFA evaluation on a private and authenticated file in the traditional two-party model, in which the file contents are known to the server. Our protocol provably achieves these properties for an arbitrarily malicious server and an honest-but-curious client, in the random oracle model.

1 Introduction

Outsourcing file storage to clouds is a dominant trend today that appears likely to continue for the foreseeable future. However, cloud storage comes with increased risks of data manipulation, since the data is stored outside the administrative control of the data owner. Numerous techniques have thus been developed to enable third parties who search on the data to confirm that the cloud service faithfully serves requests using the data owner's intended data (e.g., [26,25,21]).

Such techniques, however, typically do not account for the privacy of searches and the data itself. To protect cloud-resident files from disclosure, it is not uncommon for the data owner to encrypt her files before storing them. Specialized cryptographic protocols are then needed to permit third parties to perform searches on that data. For example, a data owner may wish to enable an antivirus vendor to perform malware scanning on her cloud-resident files without decrypting the files in the cloud. Similarly, owners of a genome database may

J. Crampton, S. Jajodia, and K. Mayes (Eds.): ESORICS 2013, LNCS 8134, pp. 147–163, 2013.

wish to enable qualified researchers to perform searches on the data (e.g., [1,2]), again without decrypting the files in the cloud. These applications are especially challenging if the third parties should be given only limited access to the data (versus disclosing all of it to them) and because the searches themselves may be sensitive: malware signatures can be used to develop malware to evade them [18,32] and searches on genome datasets may reflect proprietary research directions.

Protocols for a third-party client to perform private searches on encrypted data in the cloud, while revealing nothing to the cloud server and nothing but the search result to the client, do exist for some types of searches (e.g., [27,11,30]). To our knowledge, however, none also enforces that the cloud server employs the data that the data owner stored at the cloud server. Indeed, the traditional notion that a protocol is secure against arbitrarily malicious adversaries provides no guarantees on what *input* a malicious party may use in the protocol.

In this paper, we provide a protocol that enables a client to evaluate a deterministic finite automaton (DFA) on a file encrypted at the cloud server so that the authenticity of the file input by the server and the integrity of the computation result are both enforced. At the same time, the protocol provably protects the file contents (except for the result of the computation) from an honest-but-curious client (and heuristically from even a malicious client) and provably protects both the file contents and DFA from an arbitrarily malicious server. To our knowledge, our protocol is the first example of performing secure DFA computation on both encrypted *and authenticated* data.

Traditionally, one needs to know the file content and the signature to verify the authenticity of a file, and so the main technical difficulty in our case is to ensure computation on authenticated (signed) data without disclosing the plaintext to either party. The most common approach one might first consider to solve this problem is to leverage zero-knowledge proof techniques. By asking the data owner to publish commitments of the file character signatures, the server might then prove that his input used in the protocol is consistent with the published commitments. In the ways we see to instantiate this intuition, however, it would require much higher computation and communication costs than our protocol. Instead, we introduce a new technique to enforce correct server behavior and the authenticity of the input on which it is allowed to operate, without relying on zero-knowledge proofs at all. At a high level, the protocol takes advantage of the verifiability of the computation result to check the correctness of the server behavior. The protocol is designed so that that legitimate outputs are encoded in a small space only known to the client, and any malicious behavior by the server will result in the final output lying outside this space, which is then easily detected by the client. We prove this property (in the random oracle model) and the privacy of both the file and the DFA against an arbitrarily malicious server. We also prove the privacy of the file (except for the result of the DFA evaluation) against an honest-but-curious client.

The rest of this paper is structured as follows. We discuss related work in Section 2 and review our goals in Section 3. We detail our protocol and summarize

its security proof in Section 4. We discuss the impact of file updates in Section 5. We discuss extensions in Section 6 and conclude in Section 7.

2 Related Work

The topic on which we focus in this paper falls into the general paradigm of two-party secure computation [31,15]. The specific problem of private DFA evaluation was first studied by Troncoso-Pastoriza et al. [29] who presented a protocol for honest-but-curious adversaries in which one party can evaluate its private DFA on a string held by another party, without either party leaking any information about its input beyond what is implied by the outcome of the evaluation. Since then, the problem has been extensively studied. Frikken [13] presented a protocol that improved on the round complexity and computational costs. Gennaro et al. [14] proposed a protocol that is secure against malicious adversaries. Mohassel et al. [23] presented a protocol that significantly improves on the computational costs of both participants. Blanton and Aliasgari [4] proposed protocols that outsource the computation to two computational servers by secret sharing the DFA and data between them (with extension to multiple servers). The work by Wei and Reiter [30] is the most relevant to ours. They introduced new protocols in the cloud outsourcing scenario where a client can evaluate a DFA on the encrypted data stored on a cloud server, once authorized to do so by the data owner. However, the protocol does not guarantee the authenticity of the data input by the cloud server. The related problem of secure pattern matching has also attracted attention [16,17,19], though again without treatment of data authenticity as we consider here.

Secure computation on authenticated input was previously considered in the context of private set intersection. Several works [7,10,9,28] studied private intersection of certified sets, in which the set elements of each party must be certified by a trusted third party for use in performing the intersection. However, none considered the scenario where the data input to one party is only in ciphertext form and must remain hidden to it. In addition, to our knowledge we are the first to consider secure computation on authenticated data in the context of private DFA evaluation.

One of our protocol extensions (Section 6) secret-shares the file decryption key between the server and client in order to perform DFA evaluation on the encrypted data. In this respect, the protocols of Choi et al. [8] are related. They developed protocols based on a garbled circuit technique that enable two parties to compute any functionality after a secret decryption key is shared between them. This work, however, did not enforce authenticity of the protocol inputs.

3 Goals

A deterministic finite automaton M is a tuple $\langle Q, \Sigma, \delta, q_{\mathsf{init}} \rangle$ where Q is a set of $|Q| = n$ *states*; Σ is a set (*alphabet*) of $|\Sigma| = m$ *symbols*; $\delta : Q \times \Sigma \to Q$ is a transition function; and q_{init} is the initial state. (A DFA can also specify a

set $F \subseteq Q$ of accepting states; we ignore this here to save space, though our protocols can easily be adapted to accommodate it, similar to the techniques suggested in previous work [30].) Our goal is to enable a client holding a DFA M to interact with a server holding a file ciphertext to evaluate M on the file plaintext. More specifically, the client should output the final state to which the file plaintext drives the DFA; i.e., if the plaintext file is a sequence $\langle \sigma_k \rangle_{k \in [\ell]}$ where $[\ell]$ denotes the set $\{0, 1, \ldots, \ell - 1\}$ and where each $\sigma_k \in \Sigma$, then the client should output $\delta(\ldots \delta(\delta(q_{\mathsf{init}}, \sigma_0), \sigma_1), \ldots, \sigma_{\ell-1})$. We also permit the client to learn the file length ℓ and the server to learn the number of states n in the client's DFA. (Indeed, because the DFA output leaks $\log n$ bits about the file to the client, the server should know n to measure the leakage to the client and to limit the number of DFA queries the client is allowed, accordingly.) However, the client should learn nothing else about the file; the server should learn nothing else about the client's DFA and nothing about the file plaintext.

An additional goal of our protocols — and their main contribution over prior work — is to ensure that the client detects if the server deviates from the protocol. More specifically, we presume that a data owner stores the file ciphertext at the server, together with accompanying authentication data. We require that the client return the result of evaluating its DFA on the file stored by the data owner or else that the client detect the misbehavior of the server. In this paper we do not explicitly concern ourselves with misbehavior of the client, owing to the use cases outlined in Section 1 that involve a partially trusted third-party customer or service provider (e.g., antivirus vendor). That said, we believe our protocol to be heuristically secure against an arbitrarily malicious client.

4 Private DFA Evaluation on Signed and Encrypted Data

In this section we present a protocol meeting the goals described in Section 3: the client learns only the length of the file and the output of his DFA evaluation on the file stored at the server; the server learns only the number of states in the client's DFA and the length of the file; and the client detects any misbehavior by the server that would cause him to return an incorrect result. Again, we do not consider misbehavior of the client here; the client is honest-but-curious only. In this section we consider the file as static. The impact of file updates will be discussed in Section 5.

4.1 Preliminaries

Let "$\leftarrow$" denote assignment and "$s \xleftarrow{\$} S$" denote the assignment to s of a randomly chosen element of set S. Let κ be a security parameter. Let ParamGen be an algorithm that, on input 1^κ, produces $(p, G_1, G_2, g, e) \leftarrow \mathsf{ParamGen}(1^\kappa)$ where p is a prime; G_1 and G_2 are multiplicative groups of order p; g is a generator of G_1; and $e : G_1 \times G_1 \rightarrow G_2$ is an efficiently computable bilinear map such that $e(P^u, Q^v) = e(P, Q)^{uv}$ for any $P, Q \in G_1$ and any $u, v \in \mathbb{Z}_p^*$.

BLS Signatures. Our protocol makes use of the Boneh-Lynn-Shacham (BLS) signature scheme [6]. Suppose $(p, G_1, G_2, g, e) \leftarrow \mathsf{ParamGen}(1^\kappa)$ and let H_1 be a hash function $\mathsf{H}_1 : \{0,1\}^* \to G_1$. The BLS scheme consists of a triple of algorithms $(\mathsf{BLSKeyGen}, \mathsf{BLSSign}, \mathsf{BLSVerify})$, defined as follows.

$\mathsf{BLSKeyGen}(p, G_1, G_2, g, e)$: Select $x \xleftarrow{\$} \mathbb{Z}_p^*$. Return private signing key $\langle G_1, x\rangle$ and public verification key $\langle p, G_1, G_2, g, e, h\rangle$ where $h \leftarrow g^x$.

$\mathsf{BLSSign}_{\langle G_1, x\rangle}(m)$: Return the signature $\mathsf{H}_1(m)^x$.

$\mathsf{BLSVerify}_{\langle p, G_1, G_2, g, e, h\rangle}(m, s)$: Return true if $e(\mathsf{H}_1(m), h) = e(s, g)$ and false otherwise.

Paillier encryption. Our scheme is built using the additively homomorphic encryption scheme due to Paillier [24]. This cryptosystem has a plaintext space $\mathbb{R}$ where $\langle \mathbb{R}, +_{\mathbb{R}}, \cdot_{\mathbb{R}}\rangle$ denotes a commutative ring. Specifically, this encryption scheme includes algorithms PGen, PEnc, and PDec where: PGen is a randomized algorithm that on input 1^κ outputs a public-key/private-key pair $(pek, pdk) \leftarrow \mathsf{PGen}(1^\kappa)$; PEnc is a randomized algorithm that on input public key pek and plaintext $m \in \mathbb{R}$ (where $\mathbb{R}$ can be determined as a function of pek) produces a ciphertext $c \leftarrow \mathsf{PEnc}_{pek}(m)$, where $c \in C_{pek}$ and C_{pek} is the ciphertext space determined by pek; and PDec is a deterministic algorithm that on input a private key pdk and ciphertext $c \in C_{pek}$ produces a plaintext $m \leftarrow \mathsf{PDec}_{pdk}(c)$ where $m \in \mathbb{R}$. In addition, $\mathcal{E}$ supports an operation $+_{pek}$ on ciphertexts such that for any public-key/private-key pair (pek, pdk), $\mathsf{PDec}_{pdk}(\mathsf{PEnc}_{pek}(m_1) +_{pek} \mathsf{PEnc}_{pek}(m_2)) = m_1 +_{\mathbb{R}} m_2$. Using $+_{pek}$, it is possible to implement $\cdot_{pek}$ for which $\mathsf{PDec}_{pdk}(m_2 \cdot_{pek} \mathsf{PEnc}_{pek}(m_1)) = m_1 \cdot_{\mathbb{R}} m_2$.

In Paillier encryption, the ring $\mathbb{R}$ is $\mathbb{Z}_N$, the ciphertext space $C_{\langle N,g\rangle}$ is $\mathbb{Z}_{N^2}^*$, and the relevant algorithms are as follows.

$\mathsf{PGen}(1^\kappa)$: Choose random $\kappa/2$-bit strong primes p_1, p_2; set $N \leftarrow p_1 p_2$; choose $g \in \mathbb{Z}_{N^2}^*$ with order a multiple of N; and return the public key $\langle N, g\rangle$ and private key $\langle N, g, \lambda(N)\rangle$ where $\lambda(N)$ is the Carmichael function of N.

$\mathsf{PEnc}_{\langle N,g\rangle}(m)$: Select $r \xleftarrow{\$} \mathbb{Z}_N^*$ and return $g^m r^N \bmod N^2$.

$\mathsf{PDec}_{\langle N,g,\lambda(N)\rangle}(c)$: Return $m = \frac{L(c^{\lambda(N)} \bmod N^2)}{L(g^{\lambda(N)} \bmod N^2)} \bmod N$, where L is a function that takes input elements from the set $\{u < N^2 \mid u \equiv 1 \bmod N\}$ and returns $L(u) = \frac{u-1}{N}$.

$c_1 +_{\langle N,g\rangle} c_2$: Return $c_1 c_2 \bmod N^2$.

$m \cdot_{\langle N,g\rangle} c$: Return $c^m \bmod N^2$.

We use $\sum_{pek}$ to denote summation using $+_{pek}$; $\sum_{\mathbb{R}}$ to denote summation using $+_{\mathbb{R}}$; and $\prod_{\mathbb{R}}$ to denote the product using $\cdot_{\mathbb{R}}$ of a sequence.

4.2 Initial Construction without File Encryption

We denote the file stored at the server as consisting of characters $\sigma_0, \ldots, \sigma_{\ell-1}$, where each $\sigma_k \in \Sigma$. Prior to storing this file at the server, however, the data owner

uses its private BLS signing key $\langle G_1, x\rangle$ to produce $s_k \leftarrow \mathsf{BLSSign}_{\langle G_1,x\rangle}(\sigma_k||k)$ for each $k \in [\ell]$ — i.e., a per-file-character signature that incorporates the position of the character in the file[1] — and stores these signed characters at the server, instead. (Here, "||" denotes concatenation.) Note that since $s_k = \mathsf{H}_1(\sigma_k||k)^x$, anyone knowing the corresponding verification key $\langle p, G_1, G_2, g, e, h\rangle$ cannot only verify s_k but can also extract σ_k and k, by simply testing for each $\sigma \in \Sigma$ and $k \in [\ell]$ whether $e(\mathsf{H}_1(\sigma||k), h) = e(s_k, g)$. As such, while in our initial protocol description, the data owner stores $s_0, \ldots, s_{\ell-1}$ at the server, this implicitly conveys $\sigma_0, \ldots, \sigma_{\ell-1}$, as well.

The basic structure of the protocol, which is borrowed from previous work [30], involves the client encoding its DFA transition function δ 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. In our protocol, the client and server then evaluate this polynomial together, using a single round of interaction per state transition (i.e., per file character), in such a way that the client observes only ciphertexts of states and file characters and the server observes only a randomly blinded state. More specifically, in our protocol, if the current DFA state is q, then the server observes only $\pi(q) +_{\mathbb{R}} \varphi$ for $\varphi \xleftarrow{\$} \mathbb{R}$ chosen by the client and where $\pi : Q \to \mathbb{R}$ maps DFA states to distinct ring elements. The client, with knowledge of π and φ, can calculate $f(x, y)$ so that $f(\pi(q) +_{\mathbb{R}} \varphi, \sigma) = \pi(\delta(q, \sigma))$ for each $q \in Q$ and $\sigma \in \Sigma$. Then, starting with a ciphertext of $\pi(q)$ for the DFA state q resulting from processing file characters $\sigma_0, \ldots, \sigma_{k-1}$, the client can interact with the server to obtain a ciphertext of $f(\pi(q) +_{\mathbb{R}} \varphi, \sigma_k)$ [30].

The central innovation in our protocol is a technique by which the client, without knowing s_k, can compute an encoding of the file character σ_k that the server must use in round k of the evaluation. If the server does not, it "throws off" the evaluation in a way that the server cannot predict. As a result, if the server deviates from the protocol, the end result of the evaluation will be an unpredictable element of the ring $\mathbb{R}$, which will not correspond to *any* state of the DFA with overwhelming probability. To accomplish this, the client defines the encoding of character $\sigma \in \Sigma$ and position $k \in [\ell]$ to be $\tau(\sigma, k, \psi_k) = \mathsf{H}_2(e(\mathsf{H}_1(\sigma||k)^{\psi_k}, h))$, where H_2 is a hash function $\mathsf{H}_2 : G_2 \to \mathbb{R}$ (modeled as a random oracle) and where $\psi_k \xleftarrow{\$} \mathbb{Z}_p^*$ is selected by the client in the round for the k-th character. If the client sends $\Psi_k \leftarrow g^{\psi_k}$ to the server in the round for the k-th character, then the server can compute $\tau(\sigma_k, k, \psi_k)$ for the file character σ_k as $\tau(\sigma_k, k, \psi_k) = \mathsf{H}_2(e(s_k, \Psi_k))$. However, without ψ_k the server will be unable to compute the encoding $\tau(\sigma, k, \psi_k)$ for any $\sigma \neq \sigma_k$.

The final difficulty to overcome lies in the fact that the client, by altering the encoding of each character $\sigma \in \Sigma$ per round k, must also recompute $f(x, y)$ to account for this new encoding. As such, the client recomputes $f(x, y)$ to satisfy $f(\pi(q) +_{\mathbb{R}} \varphi_k, \tau(\sigma, k, \psi_k)) = \pi(\delta(q, \sigma))$ per round k, for every $q \in Q$ and

[1] The file name or other identifier could be included along with the character position, to detect the exchange of characters between files. Similarly, the length ℓ can be included to detect file truncation. These issues are discussed further in Section 5.

$\sigma \in \Sigma$. In our algorithm, we encapsulate this calculation as $\langle a_{ij} \rangle_{i \in [n], j \in [m]} \leftarrow \mathsf{ToPoly}(Q, \Sigma, \delta, \pi, k, \varphi_k, \beta_k, \psi_k)$ where $\langle a_{ij} \rangle_{i \in [n], j \in [m]}$ are the coefficients forming f, i.e., so that $f(x, y) = \sum_{i=0}^{n-1}{}_{\mathbb{R}} \sum_{j=0}^{m-1} a_{ij} \cdot_{\mathbb{R}} x^i \cdot_{\mathbb{R}} y^j$. (The value β_k will become relevant in Section 4.3 and can be ignored for now.)

This protocol is shown in Figure 1. The protocol is written with the steps performed by the client listed on the left (lines c101–c120), with those performed by the server on the right (lines s101–s113), and with the messages exchanged between them in the middle (lines m101–m106). The client takes as input the data owner's public verification key $\langle p, G_1, G_2, g, e, h \rangle$, a public encryption key ek', and its DFA $\langle Q, \Sigma, \delta, q_{\mathsf{init}} \rangle$. (For the moment, ignore the additional input dk, which will be discussed in Section 4.3.) The server takes as input $\langle p, G_1, G_2, g, e, h \rangle$, the DFA alphabet Σ, and the signed file characters s_0, …, $s_{\ell-1}$, i.e., signed with the data owner's private key $\langle G_1, x \rangle$ corresponding to $\langle p, G_1, G_2, g, e, h \rangle$. (Again, please ignore the b_k values for now. These will be discussed in Section 4.3.) Note that neither the client nor the server receives any information about the private key dk', and so values encrypted under ek' (θ in line c104, and ρ in line c109) are never decrypted or otherwise used in the protocol. These values are included in the protocol only to simplify its proof and need not be included in a real implementation of the protocol.

At the beginning of the protocol, the server generates the public/private key pair (pek, pdk) (line s102) that defines the ring $\mathbb{R}$ for the protocol run. The server conveys pek and the file length ℓ to the client (m101). Upon receiving this message, the client selects an injection $\pi : Q \to \mathbb{R}$ at random from the set of all such injections, denoted $\mathsf{Injs}(Q \to \mathbb{R})$ (c103). The client sends the number n of states in his DFA in message m102. (To simplify our proofs, the client also sends the chosen injection π encrypted under ek' to server, denoted by θ. We will not discuss this further here.)

The heart of the protocol is the loop represented by lines c106–c117 for the client and lines s104–s112 for the server. The client begins each iteration of this loop with a ciphertext α of the current DFA state, which it blinds with the blinding term φ_k (c107) using the additive homomorphic property of Paillier encryption (c108). The client also selects ψ_k (c110) and creates Ψ_k (c111) as described above, and sends the now-blinded ciphertext α and Ψ_k to the server (m103). After decrypting the blinded state γ (s105) and using Ψ_k and s_k to create the encoding $\eta = \tau(\sigma, k, \psi_k)$ for the character σ_k being processed in this loop iteration (s106), the server creates the encryption of $\gamma^i \cdot_{\mathbb{R}} \eta^j$ for each $i \in [n]$ and $j \in [m]$ (s107–s111). After the server sends these values back to the client (m104), the client uses them together with the coefficients of f that it computed as described above (c113) to assemble a ciphertext of the new DFA state (c116).

After this loop iterates ℓ times, the client sends the state ciphertext to the server (m105). The server decrypts the (random) state (s113) and returns it (m106). The client checks to be sure that the result represents a valid state (c118) and, if so, returns the corresponding state as the result (c120).

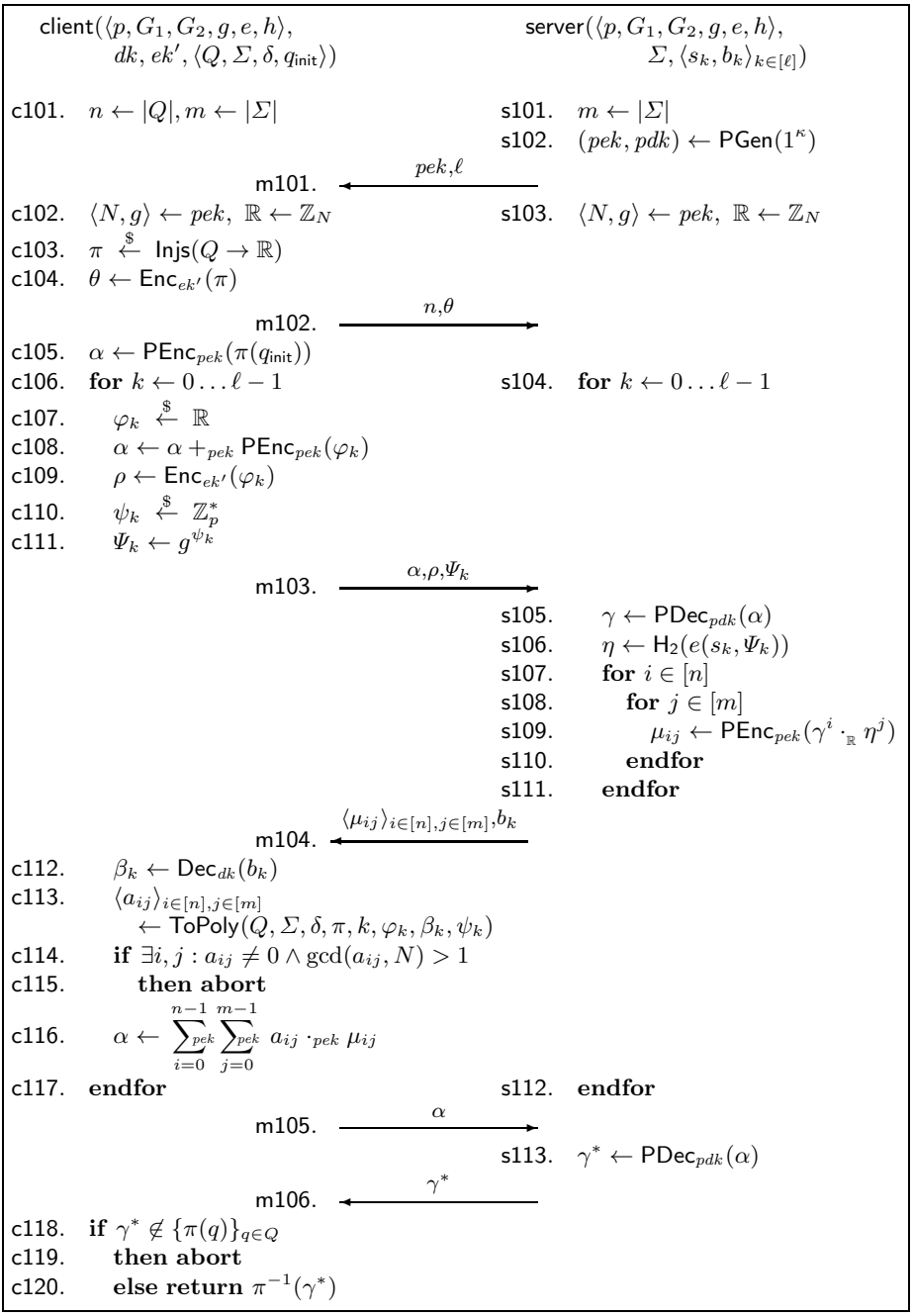

Fig. 1. Protocol Π, described in Section 4

4.3 Adding File Encryption

As presented so far, our protocol guarantees the integrity of the DFA evaluation against a malicious server. However, the confidentiality of the file content is not protected from the server because the signatures of the file characters are known to the server. With cloud outsourcing becoming increasingly popular, there is need to enable a data owner to outsource her file to the cloud while protecting its privacy, as well, against a potentially untrusted cloud provider. So, in this section, we refine our protocol so that it provides the same guarantees while also protecting the confidentiality of the file content from the server.

As we described our protocol so far, the server holds the BLS signature $s_k = \mathsf{H}_1(\sigma_k||k)^x$, which enables him to learn σ_k by testing for each $\sigma \in \Sigma$ whether $e(\mathsf{H}_1(\sigma||k), h) = e(s_k, g)$. So, to hide σ_k from the server, it is necessary to change the signature s_k to prevent the server from confirming a guess at the value of σ_k.

To do so, in our full protocol the data owner randomizes the signature by raising it to a random power, i.e., $s_k \leftarrow \mathsf{H}_1(\sigma||k)^{x \cdot \beta_k}$ where $\beta_k \xleftarrow{\$} \mathbb{Z}_p^*$. s_k then does not leak information about σ_k to the server because it is randomly distributed in G_1. However, this randomization also introduces new difficulties for the server and client to perform the DFA evaluation, since both of them need to be able to compute the same encoding for each σ_k despite s_k being randomized in this way.

To facilitate this evaluation, the data owner encrypts β_k under a public key ek of an encryption scheme whose plaintext space includes $\mathbb{Z}_p^*$ and provides its ciphertext, denoted b_k, along with s_k to the server; see the input arguments to server in Figure 1. Of course, the server should not be able to decrypt b_k, since this would again enable him to reconstruct σ_k. As such, the data owner provides the corresponding private decryption key dk only to the client; see the input arguments to the client. Analogous to previous protocols [30], conveying dk can serve as a step by which the data owner authorizes a client to perform DFA queries on its file stored at the server. (In Section 6, we summarize an alternative approach that does not disclose dk or $\langle \beta_k \rangle_{k \in [\ell]}$ to the client.)

Given this setup, the full protocol Π thus executes the following additional steps. First, the client defines the encoding of character $\sigma \in \Sigma$ and position $k \in [\ell]$ to be $\tau(\sigma, k, \beta_k, \psi_k) = \mathsf{H}_2(e(\mathsf{H}_1(\sigma||k)^{\beta_k \psi_k}, h))$, where again H_2 is a hash function $\mathsf{H}_2 : G_2 \to \mathbb{R}$ (modeled as a random oracle) and where $\psi_k \xleftarrow{\$} \mathbb{Z}_p^*$ is selected by the client in the round for character k. Note that the client needs to know β_k to compute $\tau(\sigma, k, \beta_k, \psi_k)$, and recall that the client needs to know $\tau(\sigma, k, \beta_k, \psi_k)$ for each $\sigma \in \Sigma$ in order to compute $f(x, y)$ to satisfy $f(\pi(q) +_{\mathbb{R}} \varphi_k, \tau(\sigma, k, \beta_k, \psi_k)) = \pi(\delta(q, \sigma))$ for every $q \in Q$ and $\sigma \in \Sigma$. Therefore, it is necessary for the client to include β_k as an argument to the ToPoly call (i.e., $\mathsf{ToPoly}(Q, \Sigma, \delta, \pi, k, \varphi_k, \beta_k, \psi_k)$ in c113) and to delay that call until after receiving b_k in m104 and using it to obtain β_k (c112).

4.4 Communication and Storage

Protocol Π has a communication complexity of $O(\ell mn\kappa)$ bits, dominated by message m104 consisting of mn elements of $\mathbb{Z}^*_{N^2}$ sent by the server in each of ℓ rounds, where $pek = \langle N, g\rangle$ and N is κ bits in length. The storage cost on the server is dominated by the size of $\langle s_k, b_k\rangle_{k\in[\ell]}$. Now letting κ denote the maximum of the security parameters for the BLS signatures (i.e., the s_k values) and the ciphertexts (i.e., the b_k values), and assuming that the bit length of each value type is linear in its security parameter (which is the case for BLS signatures and, say, Paillier ciphertexts), the storage cost is $O(\kappa\ell)$ bits.

4.5 Security

For brevity, we defer a full proof of security for Π to a forthcoming technical report. In this section we simply highlight the central insights and lemmas needed to complete that proof.

Privacy against server *adversaries.* The insight needed for arguing file and DFA privacy against server adversaries is to notice that, aside from $\langle b_k\rangle_{k\in[\ell]}$ provided as input to the server and the encrypted function θ sent by the client (m102), the values observed by the server are independent of the file contents or the DFA state. That is, each $s_k = \mathsf{H}_1(\sigma||k)^{x\cdot\beta_k}$ is distributed independently of σ because $\beta_k \xleftarrow{\$} \mathbb{Z}^*_p$, and the values $\gamma \leftarrow \mathsf{PDec}_{pdk}(\alpha)$ that the server recovers in line s105 are independent of the current DFA state and the file contents, owing to its blinding by the client (c107–c108). Similarly, γ^* is independent of the DFA and file contents because it is simply a random ring element determined by the random selection of π in line c103, and no other output from π is ever disclosed to the server. Also note that ρ and Ψ_k sent to the server (m103) are independent of the file characters or DFA states. Consequently, any information leakage about the file or DFA to the server must originate in a leakage either from the ciphertexts $\langle b_k\rangle_{k\in[\ell]}$ or from the ciphertext θ, for which the server holds neither decryption key. Consequently, it is possible to reduce the DFA and file privacy against server adversaries to the IND-CPA security [3] of encryption under ek or ek', respectively.

Privacy against honest-but-curious client *adversaries.* The final state γ^* of the DFA evaluation is revealed to the client in line m106, but aside from this value, the only other values sent to the client are a Paillier public key pek (m101), ciphertexts $\langle\mu_{ij}\rangle_{i\in[n],j\in[m]}$ encrypted under that public key, and the ciphertext b_k. The plaintext β_k of b_k is independent of the file content, and so its disclosure to the client (c112) does not reveal additional information about the file. Consequently, any leakage about the file (beyond the final state γ^* to which the file pushed the DFA) must originate from the ciphertexts $\langle\mu_{ij}\rangle_{i\in[n],j\in[m]}$ and so can be used to attack the IND-CPA security [3] of the Paillier encryption scheme.

This reasoning pertains equally well to malicious client-compromising adversaries and so we believe our protocol is heuristically secure against malicious

client adversaries, as well. However, the simulation for the client adversary uses the plaintexts of the values θ (m102) and ρ (m103) sent by the client, which are correct only if the client is honest-but-curious. We could *force* the correctness of these values against an arbitrarily malicious client through the addition of zero-knowledge proofs, but we do not pursue that here.

Detection of server *misbehavior.* There are essentially two avenues by which a server might attempt to misbehave while escaping detection. The first is to create $\tau(\sigma, k, \beta_k, \psi_k) = \mathsf{H}_2(e(\mathsf{H}_1(\sigma||k)^{\beta_k\psi_k}, h))$ for some $\sigma \neq \sigma_k$, and to use $\tau(\sigma, k, \beta_k, \psi_k)$ as η in the protocol. The second is to cause the client to execute a state transition into an erroneous state in Q without computing $\tau(\sigma, k, \beta_k, \psi_k)$ for some $\sigma \neq \sigma_k$. We first show that the former implies the ability to break the *bilinear computational Diffie-Hellman assumption* [6]:

Assumption 1. *For any probabilistic polynomial-time adversary $\mathcal{A}$,*

$$\mathbb{P}\left(v = e(g,g)^{z_1 z_2 z_3} \;\middle|\; \begin{array}{l}(p, G_1, G_2, g, e) \leftarrow \mathsf{ParamGen}(1^\kappa);\\ z_1, z_2, z_3 \xleftarrow{\$} \mathbb{Z}_p^*; v \leftarrow \mathcal{A}(p, G_1, G_2, g, e, g^{z_1}, g^{z_2}, g^{z_3})\end{array}\right)$$

is negligible as a function of κ.[2]

Lemma 1. *Let H_1 and H_2 be random oracles. Under Assumption 1, there is no probabilistic polynomial time* server*-compromising adversary $\mathcal{S}$ that computes $\tau(\sigma, k, \beta_k, \psi_k)$ for some $k \in [\ell]$ and $\sigma \neq \sigma_k$ with non-negligible probability, after interacting with the* client *in protocol Π.*

Proof. Suppose such a server adversary $\mathcal{S}$ exists. We build an adversary $\mathcal{A}$ that takes in a challenge $(p, G_1, G_2, g, e, g^{z_1}, g^{z_2}, g^{z_3})$ as input, interacts with $\mathcal{S}$, and outputs $e(g,g)^{z_1 z_2 z_3}$ with non-negligible probability, violating Assumption 1. $\mathcal{A}$ is defined as follows, where $Z_1 = g^{z_1}$, $Z_2 = g^{z_2}$ and $Z_3 = g^{z_3}$:

- **Setup:** $\mathcal{A}$ generates a public/private key pair (ek, dk) for an encryption scheme, a file length $\ell > 0$, an alphabet Σ such that $|\Sigma| > 1$, and a sequence of plaintext file characters $\langle\sigma_k\rangle_{k\in[\ell]}$, $\sigma_k \in \Sigma$. $\mathcal{A}$ sets $\mathsf{H}_1(\sigma_k||k) \leftarrow g^u$ where $u \xleftarrow{\$} \mathbb{Z}_p^*$ and then computes the encrypted file sequence $\langle s_k, b_k\rangle_{k\in[\ell]}$ such that $s_k \leftarrow Z_1^{u\beta_k}$ for $\beta_k \xleftarrow{\$} \mathbb{Z}_p^*$ and $b_k \leftarrow \mathsf{Enc}_{ek}(\beta_k)$. $\mathcal{A}$ invokes $\mathcal{S}(\langle p, G_1, G_2, g, e, Z_1\rangle, \Sigma, \langle s_k, b_k\rangle_{k\in[\ell]})$. Note that the file ciphertext $\langle s_k, b_k\rangle_{k\in[\ell]}$ is well formed because $e(s_k, g) = e(Z_1^{u\beta_k}, g) = e(g^{z_1 u\beta_k}, g) = e(g,g)^{z_1 u\beta_k} = e(g^u, g^{z_1})^{\beta_k} = e(\mathsf{H}_1(\sigma_k||k), Z_1)^{\beta_k}$, as in the real protocol. Finally, $\mathcal{A}$ chooses $k^* \xleftarrow{\$} [\ell]$ and $\sigma^* \xleftarrow{\$} \Sigma \setminus \{\sigma_{k^*}\}$.
- **Simulation for $\mathcal{S}$:** After receiving *pek* and ℓ from $\mathcal{S}$ (m101), $\mathcal{A}$ chooses $n > 0$ arbitrarily and computes θ exactly as in the real protocol, using an encryption key ek' of its own choosing. $\mathcal{A}$ sends n and θ to $\mathcal{S}$ (m102).

[2] A function μ is *negligible as a function of κ* if for every positive polynomial p, there is some κ_0 such that $\mu(\kappa) < 1/p(\kappa)$ for all $\kappa > \kappa_0$.

In round $k \in [\ell]$, $\mathcal{A}$ computes α to be the ciphertext of random element of $\mathbb{R}$. If $k \neq k^*$, then $\mathcal{A}$ generates the random challenge Ψ_k exactly as specified in c110–c111. If $k = k^*$, then $\mathcal{A}$ sets $\Psi_k \leftarrow Z_3$. In either case, $\mathcal{A}$ then sends α and Ψ_k to $\mathcal{S}$ (m103).

After ℓ such rounds, $\mathcal{A}$ computes α to be the ciphertext of a random element of $\mathbb{R}$, and sends it to $\mathcal{S}$ (m105).

- **Hash queries to H_1:** For any query that was previously posed to H_1, $\mathcal{A}$ returns the value returned to that previous query, and for new queries, $\mathcal{A}$ generates a return value as follows. If the query is $\sigma^*||k^*$, then $\mathcal{A}$ returns Z_2. For all other queries, $\mathcal{A}$ picks $u \xleftarrow{\$} \mathbb{Z}_p^*$ and returns g^u.
- **Hash queries to H_2:** For any query that was previously posed to H_2, $\mathcal{A}$ returns the value returned to that previous query. For new queries, $\mathcal{A}$ picks $r \xleftarrow{\$} \mathbb{Z}_N$ and returns r to $\mathcal{S}$.

The view that $\mathcal{A}$ simulates for $\mathcal{S}$ is indistinguishable from a real protocol execution. If $\mathcal{S}$ computes

$$\begin{aligned}\tau(\sigma^*, k^*, \beta_{k^*}, \psi_k) &= \mathsf{H}_2(e(\mathsf{H}_1(\sigma^*||k^*)^{\beta_{k^*}\psi_k}, Z_1)) \\ &= \mathsf{H}_2(e(Z_2^{\beta_{k^*}z_3}, Z_1)) \\ &= \mathsf{H}_2(e(g,g)^{z_1 z_2 z_3 \beta_{k^*}})\end{aligned}$$

then $\mathcal{A}$ can output $e(g,g)^{z_1z_2z_3}$ with non-negligible probability by selecting a random query χ that $\mathcal{S}$ made of H_2 and returning $\chi^{\beta_{k^*}^{-1} \bmod p}$. The probability that $\mathcal{A}$ outputs $e(g,g)^{z_1z_2z_3}$ is then $\frac{1}{(m-1)\cdot\ell\cdot\#(\mathsf{H}_2)}$ times the probability that $\mathcal{S}$ produces $\tau(\sigma, k, \beta_k, \psi_k)$ for some $k \in [\ell]$ and $\sigma \neq \sigma_k$, where $\#(\mathsf{H}_2)$ is the number of queries that $\mathcal{S}$ poses to H_2. If the latter probability is non-negligible, then the former is, too. □

We now consider the second possibility, i.e., that the server causes the client to execute a state transition into an erroneous state in Q without computing $\tau(\sigma, k, \beta_k, \psi_k)$ for some $\sigma \neq \sigma_k$. To prove that this happens with negligible probability, we leverage properties specific to the Paillier cryptosystem.

Lemma 2. *Let H_2 be a random oracle, and let $\mathcal{S}$ be a* server*-compromising adversary. If in no round k does $\mathcal{S}$ compute $\tau(\sigma, k, \beta_k, \psi_k)$ for some $\sigma \neq \sigma_k$, then the* client *outputs an incorrect state $q \in Q$ with probability at most negligibly more than $\frac{n-1}{N}$.*

Proof. In round k, the client transitions to the next DFA state by encoding the DFA transition function using a polynomial f satisfying $f(\pi(q) +_{\mathbb{R}} \varphi_k, \tau(\sigma, k, \beta_k, \psi_k)) = \pi(\delta(q, \sigma))$ for every $q \in Q$ and $\sigma \in \Sigma$; let $f(x,y) = \sum_{i=0}^{n-1}{}_{\mathbb{R}} \sum_{j=0}^{m-1} a_{ij} \cdot_{\mathbb{R}} x^i \cdot_{\mathbb{R}} y^j$. To cause a state transition to an erroneous state $q' \in Q$, a server adversary must therefore produce ciphertexts $\langle \mu_{ij} \rangle_{i\in[n], j\in[m]}$ with corresponding plaintexts $\langle \nu_{ij} \rangle_{i\in[n], j\in[m]}$ so that

$$\pi(q') = \sum_{i=0}^{n-1}{}_{\mathbb{R}} \sum_{j=0}^{m-1}{}_{\mathbb{R}} a_{ij} \cdot_{\mathbb{R}} \nu_{ij} \tag{1}$$

without having any information about $\tau(\sigma, k, \beta_k, \psi_k)$ for any $\sigma \neq \sigma_k$ (since H_2 is a random oracle). Note that the distribution of $\langle a_{ij} \rangle_{i \in [n], j \in [m]}$ is *not* independent of the DFA transition function δ and the injection π. That is, once π is fixed, only certain values for $\langle a_{ij} \rangle_{i \in [n], j \in [m]}$ are possible.

We argue the result under the conservative assumption that δ and π *uniquely determine* $\langle a_{ij} \rangle_{i \in [n], j \in [m]}$ (which in general they do not). Even then, for any $i' \in [n]$ and $j' \in [m]$ such that $a_{i'j'} \neq 0$ and $\gcd(a_{i'j'}, N) = 1$ (lines c114–c115 abort the protocol if $\gcd(a_{ij}, N) > 1$ for some $a_{ij} \neq 0$), and for any choices of $\langle \nu_{ij} \rangle_{i \in [n], j \in [m]}$ excepting $\nu_{i'j'}$, there is exactly one value for $\nu_{i'j'}$ in $\mathbb{Z}_N$ that satisfies (1). Moreover, prior to the last message sent by the client (m105), the server receives no information about π. So, the probability $\mathcal{S}$ succeeds in selecting $\langle \nu_{ij} \rangle_{i \in [n], j \in [m]}$ to satisfy (1) is $\frac{1}{N}$, and since there are $n - 1$ possible erroneous states q', the probability $\mathcal{S}$ succeeds in causing an erroneous state transition to any $q' \in Q$ is at most $\frac{n-1}{N}$.

Finally, while the server learns $\pi(q)$ for one $q \in Q$ in the last client-to-server message (m105) — if it behaved thus far — it does so only for the correct state q at this point. Again, it can then guess $\pi(q')$ for an incorrect $q' \in Q$ to return as γ^* with probability only $\frac{n-1}{N}$. □

5 On File Updates

Protocol Π is presented for a static file, and so in this section we consider the impact of file updates. As we discuss below, these impacts are nontrivial, and so our protocol is arguably most useful for static files.

To enable protocol Π, the data owner signs the file position k along with σ_k when producing s_k to detect the server reordering file characters, i.e., $s_k \leftarrow \mathsf{H}_1(\sigma||k)^{x \cdot \beta_k}$ where $\beta_k \xleftarrow{\$} \mathbb{Z}_p^*$. Such a representation would require any character insertion or deletion at position k to further require updating the signature $s_{k'}$ for all $k' > k$. If the total file length ℓ is also included as an input to H_1 to detect file truncation, then insertions and deletions may require updating the signatures $s_{k'}$ for all $k' < k$, as well. This latter cost can be eliminated by not including ℓ as an input to H_1 but rather to have the data owner sign ℓ and the server to forward this signature along with ℓ to the client in message m101. The former cost can be mitigated somewhat by breaking each file into blocks (essentially smaller files) so that insertions and deletions require only the affected blocks to be rewritten. In this case, the block index within the file should presumably also be included as an input to H_1 to detect block reorderings by the server.

Even with these modifications, there remain other complexities in handling file updates, in that a server could simply use a stale version of the file when performing protocol Π with the client, ignoring any earlier updates to the file by the data owner. Detecting a server that selectively suppresses updates seems to require additional interaction between the data owner and the client and has been the subject of much study (for file stores subject to reads and updates only) under the banner of *fork consistency* [22]. We leave as future work the

integration of our DFA evaluation techniques with these ideas, i.e., so that DFA evaluations performed against stale files are efficiently detected when the client subsequently interacts with the data owner.

6 Extensions

The protocol Π can be extended in various ways that may be of interest and that we will discuss here. The first "extension" is simply the removal of the file encryption step described in Section 4.3, which is suitable for the standard two-party model where the server's input need not be kept secret from the server himself. This simplification eliminates the dk, β_k and b_k values from the protocol, implicitly setting $\beta_k = 1$.

A more interesting variant of the protocol addresses the concern that the protocol as stated in Figure 1 discloses the decryption key dk and the values $\langle \beta_k \rangle_{k \in [\ell]}$ to the client, either of which can be used to decrypt the file from its ciphertext $\langle s_k, b_k \rangle_{k \in [\ell]}$. While this file ciphertext is not disclosed to the client during the protocol, it seems unnecessarily permissive to disclose its decryption key to every client that performs a DFA evaluation on the file: if the file ciphertext were ever unintentionally disclosed, then any such client could decrypt the file if it retained the key. In the rest of this section we discuss an extension to the protocol in Figure 1 to avoid disclosing dk and the values $\langle \beta_k \rangle_{k \in [\ell]}$ to the client.

In order to avoid disclosing dk to the client, one alternative is for the data owner to provide shares of dk to both the client and the server, so as to enable a two-party decryption of each b_k. Then, rather than sending only b_k to the client in message m104, the server can also send its contribution to the decryption of b_k, enabling the client to complete the decryption of b_k without learning dk itself.

Still, however, this alternative would disclose β_k to the client, which would enable it to determine σ_k if s_k were ever disclosed. To avoid disclosing β_k, one strategy is for the server to first blind β_k with another random value t_k, i.e., to execute the protocol with $\beta_k t_k$ in place of just β_k. Of course, this factor t_k would also then need to be reflected in k-th file character used in the protocol, i.e., so the server would use $s_k^{t_k} = \mathsf{H}_1(\sigma_k || k)^{x \beta_k t_k}$ in place of s_k in the protocol. Because the server does not have access to β_k but rather has access only to its ciphertext b_k, it is necessary that the encryption scheme used to construct b_k enable the computation of a ciphertext $\hat{b}_k$ from b_k and t_k such that $\mathsf{Dec}_{dk}(\hat{b}_k) = \beta_k t_k \bmod N'$ for some value N' such that $p \mid N'$. In this case, selecting $t_k \stackrel{\$}{\leftarrow} \mathbb{Z}_{N'}$ suffices to ensure that $\beta_k t_k \bmod N'$ is distributed independently of β_k and so hides β_k from the client when it learns $\beta_k t_k \bmod N'$.

An encryption scheme meeting our requirements (supporting two-party decryption and homomorphism on ciphertexts) is ElGamal encryption [12] in a subgroup of $\mathbb{Z}_{N'}^*$. However, note that setting $N' = p$ is inefficient: the security parameter κ and so the size of p required for security is an order of magnitude less for BLS signing than it would be for ElGamal encryption in a subgroup of $\mathbb{Z}_p^*$ [20], and so setting $N' = p$ would add considerable expense to the protocol.

As such, a more efficient construction would be to choose $N' = pp'$ for another prime p'. ElGamal encryption is believed to be secure with a composite modulus even if its factorization is known [5].

7 Conclusion

We presented a protocol by which a data owner can outsource storage of a file to an untrusted cloud server while still enabling partially trusted third-party clients (e.g., customers and service providers) to evaluate DFAs on that data. Our protocol is novel in provably enabling the client to detect the server's misbehavior — including the use of a file other than the data owner's in the protocol — in the random oracle model, while simultaneously protecting the privacy of the file and of the DFA from an arbitrarily malicious server. Moreover, our protocol provably protects the privacy of the file (except for the DFA evaluation result) from an honest-but-curious client (and heuristically does so from an arbitrarily malicious one). We accomplish these goals without the use of zero-knowledge proofs, yielding a protocol that is more efficient than alternatives of which we are aware. We believe that our protocol has applications to malware scanning or genome analysis on encrypted, cloud-resident data, and we plan to explore these applications in ongoing work.

Acknowledgments. This work was supported in part by NSF grant 0910483.

References

1. GenBank, `http://www.ncbi.nlm.nih.gov/genbank/`
2. United Kingdom National DNA Database, `http://www.npia.police.uk/en/8934.htm`
3. Bellare, M., Desai, A., Pointcheval, D., Rogaway, P.: Relations among notions of security for public-key encryption schemes. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 26–45. Springer, Heidelberg (1998)
4. Blanton, M., Aliasgari, M.: Secure outsourcing of DNA searching via finite automata. In: Foresti, S., Jajodia, S. (eds.) Data and Applications Security and Privacy XXIV. LNCS, vol. 6166, pp. 49–64. Springer, Heidelberg (2010)
5. Boneh, D.: The decision Diffie-Hellman problem. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 48–63. Springer, Heidelberg (1998)
6. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 514–532. Springer, Heidelberg (2001)
7. Camenisch, J., Zaverucha, G.M.: Private intersection of certified sets. In: Dingledine, R., Golle, P. (eds.) FC 2009. LNCS, vol. 5628, pp. 108–127. Springer, Heidelberg (2009)
8. Choi, S.G., Elbaz, A., Juels, A., Malkin, T., Yung, M.: Two-party computing with encrypted data. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 298–314. Springer, Heidelberg (2007)
9. De Cristofaro, E., Kim, J., Tsudik, G.: Linear-complexity private set intersection protocols secure in malicious model. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 213–231. Springer, Heidelberg (2010)

10. De Cristofaro, E., Tsudik, G.: Practical private set intersection protocols with linear complexity. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 143–159. Springer, Heidelberg (2010)
11. 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)
12. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31(4), 469–472 (1985)
13. Frikken, K.B.: Practical private DNA string searching and matching through efficient oblivious automata evaluation. In: Gudes, E., Vaidya, J. (eds.) Data and Applications Security XXIII. LNCS, vol. 5645, pp. 81–94. Springer, Heidelberg (2009)
14. Gennaro, R., Hazay, C., Sorensen, J.S.: Text search protocols with simulation based security. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 332–350. Springer, Heidelberg (2010)
15. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: 19th ACM Symposium on Theory of Computing, pp. 218–229 (1987)
16. Hazay, C., Lindell, Y.: Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. Journal of Cryptology 23(3), 422–456 (2010)
17. Hazay, C., Toft, T.: Computationally secure pattern matching in the presence of malicious adversaries. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 195–212. Springer, Heidelberg (2010)
18. Higgins, K.J.: Black hat: How to hack IPS signatures (2007), `http://www.darkreading.com/security/perimeter-security/208804656/black-hat-how-to-hack-ips-signatures.html`
19. Katz, J., Malka, L.: Secure text processing with applications to private DNA matching. In: 17th ACM Conference on Computer and Communications Security, pp. 485–492 (2010)
20. Lenstra, A.K., Verheul, E.R.: Selecting cryptographic key sizes. Journal of Cryptology 14(4), 255–293 (2001)
21. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Authenticated index structures for aggregation queries. ACM Transactions on Information and System Security 13(4) (December 2010)
22. Mazières, D., Shasha, D.: Building secure file systems out of Byzantine storage. In: 21st Symposium on Principles of Distributed Computing, pp. 108–117 (July 2002)
23. Mohassel, P., Niksefat, S., Sadeghian, S., Sadeghiyan, B.: An efficient protocol for oblivious DFA evaluation and applications. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 398–415. Springer, Heidelberg (2012)
24. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
25. Pang, H., Zhang, J., Mouratidis, K.: Scalable verification for outsourced dynamic databases. In: 35th International Conference on Very Large Databases, pp. 802–813 (2009)
26. Papamanthou, C., Tamassia, R., Triandopoulos, R.: Authenticated hash tables. In: 15th ACM Conference on Computer and Communications Security, pp. 437–448 (2008)
27. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: 2000 IEEE Symposium on Security and Privacy (2000)

28. Stefanov, E., Shi, E., Song, D.: Policy-enhanced private set intersection: sharing information while enforcing privacy policies. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) PKC 2012. LNCS, vol. 7293, pp. 413–430. Springer, Heidelberg (2012)
29. 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)
30. Wei, L., Reiter, M.K.: Third-party private DFA evaluation on encrypted files in the cloud. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 523–540. Springer, Heidelberg (2012)
31. Yao, A.C.: Protocols for secure computations. In: 23rd IEEE Symposium on Foundations of Computer Science, pp. 160–164 (1982)
32. Zhuge, J., Holz, T., Song, C., Guo, J., Han, X., Zou, W.: Studying malicious websites and the underground economy on the Chinese web. In: Workshop on the Economics of Information Security (June 2008)