Lightweight Query Authentication on Streams

STAVROS PAPADOPOULOS, Hong Kong University of Science and Technology
GRAHAM CORMODE, University of Warwick
ANTONIOS DELIGIANNAKIS and MINOS GAROFALAKIS, Technical University of Crete

We consider a *stream outsourcing* setting, where a data owner delegates the management of a set of disjoint data streams to an untrusted server. The owner authenticates his streams via signatures. The server processes continuous queries on the union of the streams for clients trusted by the owner. Along with the results, the server sends proofs of result correctness derived from the owner's signatures, which are verifiable by the clients. We design novel constructions for a collection of fundamental problems over streams represented as linear algebraic queries. In particular, our basic schemes authenticate dynamic vector sums, matrix products, and dot products. These techniques can be adapted for authenticating a wide range of important operations in streaming environments, including group-by queries, joins, in-network aggregation, similarity matching, and event processing. We also present extensions to address the case of sliding window queries, and when multiple clients are interested in different subsets of the data. These methods take advantage of a novel nonce chaining technique that we introduce, which is used to reduce the verification cost without affecting any other costs. All our schemes are lightweight and offer strong cryptographic guarantees derived from formal definitions and proofs. We experimentally confirm the practicality of our schemes in the performance-sensitive streaming setting.

Categories and Subject Descriptors: D.4.6 [**Security and Protection**]: Authentication

General Terms: Algorithms, Security

Additional Key Words and Phrases: Data integrity, data streams, query authentication

ACM Reference Format:
Stavros Papadopoulos, Graham Cormode, Antonios Deligiannakis, and Minos Garofalakis. 2014. Lightweight query authentication on streams. ACM Trans. Datab. Syst. 39, 4, Article 30 (December 2014), 45 pages.
DOI: http://dx.doi.org/10.1145/2656336

1. INTRODUCTION

Tremendous amounts of data are being generated in a streaming fashion in a variety of applications, such as Web and telephony networks, wireless sensor networks, social networks, and more. The continuous nature of such data has motivated the need for sophisticated Data Stream Management Systems (DSMSs) that offer efficient storage and reliable querying services to clients. Following research prototypes such as Stream [Arasu et al. 2003] and Aurora [Abadi et al. 2003], robust DSMSs have been deployed for many applications, including IBM's InfoSphere Streams [Nasgaard et al. 2009], Microsoft's StreamInsight [Microsoft 2010], and AT&T's Gigascope [Cranor et al. 2003]. Due to the overwhelming volume of streaming data, companies may neither possess nor wish to acquire the resources for deploying a DSMS. A practical alternative

A. Deligiannakis and M. Garofalakis were supported by the European Commission under ICT-FP7-LEADS-318809 (Large-Scale Elastic Architecture for Data-as-a-Service).
Authors' addresses: S. Papadopoulos (corresponding author, current address), Intel Labs and CSAIL, MIT, 32 Vassar Street, Cambridge, MA 02139; email: stavrosp@csail.mit.edu; G. Cormode, Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK; A. Deligiannakis and M. Garofalakis, School of Electronic and Computer Engineering, Technical University of Crete, Chania 731 00, Greece.

DOI: http://dx.doi.org/10.1145/2656336

is to outsource the stream storage and processing to a specialized third party with strong DSMS infrastructure. Outsourcing offers significant cost savings to companies, especially start-ups.

Despite its merits, outsourcing naturally raises the issue of *trust*. Specifically, the third party may act maliciously to increase profit, for example, it may collude with rival companies and present fraudulent results to bias the competition, or it may shed some of the workload and only compute on a sample of the input to save effort. Even when the server is honest, problems can arise, as it may run buggy software, or (given the scale of the problems considered) suffer equipment failure or read/write errors. It is therefore particularly important to adopt methods for *stream authentication*. These enable the clients to verify the *correctness* of the streaming results they receive from the server, that is, that they are untampered with (*integrity*), include all tuples from the data owner (*completeness*) and are up-to-date (*freshness*). The goal is to make stream authentication a lightweight operation for all parties involved, and to establish it as a standard tool for error checking in a similar way to the ubiquitous use of checksums for reliable file transfer.

Targeted problem and motivation. We consider that an *owner* possesses a set of machines $M_1, M_2, \ldots, M_m$, each generating or observing a data stream $\mathcal{X}_i$. A *client* (who is trusted by the owner) issues a continuous (i.e., long-running) query Q on the union of the streams and wishes to receive the result of Q at regular time intervals, demarcated into epochs τ. In this work, we focus on three linear algebraic queries: *vector sums*, *matrix products*, and *dot products*. In vector sums, each machine dynamically updates (through its stream) an n-element vector, and the query asks for the sum of all the m machine vectors. In matrix product, there are two machines ($m = 2$), where the first dynamically updates an $n_a \times n$ matrix, and the second an $n \times n_b$ matrix. The query asks for the $n_a \times n_b$ product of the two matrices. Finally, in dot product, there are two machines, each dynamically updating an n-element vector. The result is the dot product between the two vectors.

Instead of having the machines transmit their raw streams directly to the client and the client process manually the query on the union of the streams, we involve a third-party *server* that acts as an intermediary between the client and the machines. Specifically, the server gathers the data streams from the machines, processes the query every epoch, and returns the result to the client. The advantages of outsourcing the maintenance of the streams and query processing to the server are manyfold: (i) The server may provide a strong network infrastructure to connect to all the machines, especially if the machines are geographically remote sensors with limited wireless range. (ii) The server may be a highly scalable cloud service that can offer storage to handle the huge volume of streams. (iii) The queries in our setting are performed on a per-epoch basis, where each epoch may receive numerous updates. Instead of bombarding the client with every single update (in some verifiable manner, e.g., by authenticating them via MACs), the server can gather all updates per epoch, and produce and transmit a single result to the client in this epoch. In other words, the server can alleviate a considerable communication overhead from the client, which is an important cost especially if the client is a mobile device with battery constraints. (iv) Most importantly, the server may possess highly specialized and sophisticated tools for executing the queries very efficiently. For instance, matrix products are frequent queries in the emerging scientific databases, such as SciDB [Brown 2010]. These are highly scalable systems that deploy a cloud architecture and integrate state-of-the-art math libraries (e.g., ScaLAPACK[1]) that can efficiently process matrix products in a

[1]http://www.netlib.org/scalapack/.

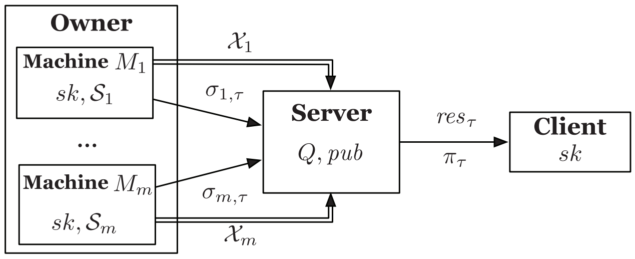


Fig. 1. System setting.

distributed manner across numerous nodes. These systems may be too expensive for the client to acquire. Moreover, note that we target settings where the client can be as simple as a smartphone device, rather than a cloud service.

Nevertheless, the main downside of outsourcing (as mentioned earlier) is that the server is untrustworthy. Towards this end, we propose the system architecture illustrated in Figure 1. Each machine M_i maintains a small *summary* $\mathcal{S}_i$ on its stream, which is updated with every new tuple arrival. At the end of every epoch τ, M_i computes a *signature* $\sigma_{i,\tau}$ on $\mathcal{S}_i$, and sends it to the server. This signature authenticates M_i's stream at that particular epoch and is created with a secret key *sk* installed by the owner at the machine. The server then processes query Q and transmits result res_τ along with a small *proof* π_τ. The proof is produced in a query-specific fashion by combining all the signatures with some public information *pub* (registered at the server by the owner at an offline setup stage). We assume the client is trusted by the owner and thus possesses *sk*. Using this key and π_τ, the client can verify the correctness of the result received for epoch τ.

Our aim is to provide the aforesaid functionality, offering *cryptographic* security and satisfying certain *performance desiderata*. In particular, our goals are to minimize the memory, communication, and computational costs for the owner and clients. This is particularly crucial in applications such as wireless sensor networks, where the owner's machines are motes with scarce resources and limited battery. The lifetime of these systems is diminished by intense operations and communication. Secondarily, we further aim to ensure that the server's costs are low.

Our contributions. The existing literature on stream authentication is limited in its applicability for a variety of reasons. First, the range of supported queries is somewhat narrow; prior work has been primarily concerned with authenticating particular computations such as `group-by`, `sum` queries that, while fundamental, do not cover all stream outsourcing scenarios. Second, the authentication cost at the owner is nontrivial; it typically entails expensive cryptographic operations (e.g., modular exponentiations) for each epoch at the owner. While the cost of one such operation is minor, the overhead imposed for high-speed data streams and short epochs can become intolerably high, especially when each machine M_i might be a low-powered embedded device. This fact also limits the data rates that the owner can process.

In contrast to current literature, we seek more general solutions that impose a minimal, essentially negligible, cost to the data owner. We first devise constructions for fundamental problems represented as linear algebraic queries. We then use these schemes as building blocks in the design of authentication techniques for a wide range of important queries in streaming environments. In more detail, our contributions are summarized as follows.

—We introduce constructions for authenticating: (i) *sums* of dynamic vectors produced by one or multiple streams, (ii) *dot products* of dynamic vectors produced by different

streams, and (iii) *products* between *dynamic matrices* generated by different streams. Our schemes are lightweight for the owner, as they mainly involve inexpensive hash operations and modular additions/multiplications in a very small finite field. They are also inexpensive for the client, who verifies the result without adding substantially to the cost of reading the output. Moreover, they impose only a small extra overhead to the computation cost of the server.
—We introduce a novel *nonce chaining* technique, that is used to optimize the verification cost of our algorithms.
—We extend our basic constructions to the case of sliding window queries and present solutions for a variety of sliding window query types. We also design further extensions for handling multiple clients that register different subset queries.
—We provide strong cryptographic guarantees for all our constructions, derived from formal definitions and proofs.
—We show how to adapt the basic schemes in order to solve a wide range of database queries in stream authentication, including `group-by` queries, joins, in-network aggregation, similarity matching, and event processing. To our knowledge, we are the first to address result authentication for such a large set of complex queries.

Roadmap. Section 2 includes necessary preliminary information and surveys the related work. Section 3 formulates the framework within which our stream authentication protocols will operate. Section 4 presents the basic constructions for authenticating the three fundamental linear algebraic queries. Section 5 first introduces a novel nonce chaining technique and then introduces optimized versions of our constructions that seek to minimize the verification cost. Section 6 extends our basic constructions to the scenario of sliding window queries. Section 7 describes extensions of our techniques to the case of multiple subset queries. Section 8 adapts our main schemes to a variety of important database applications in the stream authentication setting. Section 9 contains our experimental evaluation, and Section 10 concludes our article with future research directions.

2. BACKGROUND

Section 2.1 contains preliminary information and Section 2.2 surveys the related work on the topic.

2.1. Preliminaries

Stream model and notation. The time domain is decomposed into intervals, called *epochs*. An epoch can be perceived as a discrete timestamp denoted by τ. We assume that the clocks of the owner's machines, the server, and the client are (at least loosely) *synchronized*. This requirement is inherent in most streaming applications (e.g., sensor networks) and is orthogonal to our work. Table I summarizes the most important notation used in this article.

Adversary. Henceforth, any reference to an adversary implies a probabilistic adversary that runs in time polynomial in some security parameter s.

Negligible functions. We call a function $\nu : \mathbb{N} \rightarrow \mathbb{N}$ *negligible* in s if $\nu(s) < 1/\mathsf{poly}(s)$ for every positive polynomial $\mathsf{poly}(\cdot)$ and sufficiently large, and denote it by $\mathsf{negl}(s)$.

Pseudorandom functions. Let $F : \mathcal{K} \times S_1 \rightarrow S_2$ be an efficient, keyed function where $\mathcal{K}$, S_1, and S_2 are indexed by a security parameter s. We say that F is a *pseudorandom function* (PRF) if, for all adversaries $\mathcal{A}$, it holds that

$$|\Pr[\mathcal{A}^{F_k(\cdot)}(1^s) = 1] - \Pr[\mathcal{A}^{f(\cdot)}(1^s) = 1]| = \mathsf{negl}(s),$$

Table I. Notation

Symbol	Definition
m	Number of owner machines
M_i	Owner machine i
$\mathcal{X}_i$	The stream of tuples generated at M_i
$\mathcal{X}_i(\tau)$	The tuple sequence of M_i at or before epoch τ
Q	The continuous query of the client
$Q(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\})$	Result of Q on streams $\mathcal{X}_1, \ldots, \mathcal{X}_m$ at epoch τ
res_τ	Result sent by the server to the client at epoch τ
$\sigma_{i,\tau}$	Signature created by machine M_i at epoch τ
π_τ	Proof transmitted by the server to the client at τ
$r_{a,\tau}$, $\rho_{a,\tau}$	Key values (nonces) computed for machine a at τ
$\mathcal{S}_i$	The summary maintained at M_i at all times
pub	Public information output by the owner during setup
$\mathbf{a}, \mathbf{b}$	Symbols (in lowercase bold letters) of vectors
$\mathbf{A}, \mathbf{B}$	Symbols (in uppercase bold letters) of matrices
$\mathbf{a}[i_l : i_u]$	Subvector of $\mathbf{a}$ comprised of elements with indices $i_l, i_l + 1, \ldots, i_u$
$\mathbf{A}[i_l : i_u, j_l : j_u]$	Submatrix of $\mathbf{A}$ with rows $i_l, i_l + 1, \ldots, i_u$ and columns $j_l, j_l + 1, \ldots, j_u$
$x \overset{\$}{\leftarrow} S$	An element x being sampled uniformly from set S
$x \leftarrow \mathcal{A}$	The output x of a *probabilistic* algorithm $\mathcal{A}$
$x := \mathcal{B}$	The output x of a *deterministic* algorithm $\mathcal{B}$
'$\|\|$'	Symbol denoting string concatenation
'$\|$'	Symbol denoting logical OR
s	The security parameter
poly(s) / negl(s)	A positive polynomial in s / A negligible function in s
$[n]$	The set $\{1, 2, \ldots, n\}$
$F_k(x) \overset{\text{def}}{=} F(k, x)$	Pseudorandom function F of key k and message x
sk	The secret key of the owner
p	A prime number with bit size $\Theta(s)$
$\mathbb{Z}_p$ / $\mathbb{G}$	The finite field / cyclic group our algorithms operate on

where $F_k(x) \overset{\text{def}}{=} F(k, x)$, $k \overset{\$}{\leftarrow} \mathcal{K}$ and $f \overset{\$}{\leftarrow} (S_1 \rightarrow S_2)$. Simply stated, an adversary distinguishes a PRF from a truly random function only with negligible probability in s.

Cyclic groups, generators and multiplicative cyclic groups [Menezes et al. 1996]. Let $\mathbb{G}$ be a group, let $p = |\mathbb{G}|$ denote the order of $\mathbb{G}$, and let $\mathbf{1}$ represent the identity element of $\mathbb{G}$. For any element $g \in \mathbb{G}$, the order of g is the smallest positive integer n such that $g^n = \mathbf{1}$. Let $\langle g \rangle = \{g^i : i \in \mathbb{Z}_n\} = \{g^0, g^1, \ldots, g^{n-1}\}$ denote the set of group elements generated by g. The group $\mathbb{G}$ is called *cyclic* if there exists an element $g \in \mathbb{G}$ such that $\langle g \rangle = \mathbb{G}$. In such a case, g is called a *generator* of $\mathbb{G}$. A cyclic group $\mathbb{G}$ with the binary operator of multiplication is called a *multiplicative* cyclic group.

The Diffie-Hellman Exponent (n-DHE) Assumption [Camenisch et al. 2009]. Our security relies on a variant of the well-known discrete logarithm problem. Let $\mathbb{G}$ be a multiplicative cyclic group of order p, $g \in \mathbb{G}$ a generator of $\mathbb{G}$, and s the bit size of p. The n-DHE problem is defined as follows: given set $\mathcal{V} = \{g, g^k, g^{k^2}, \ldots, g^{k^n}, g^{k^{n+2}}, \ldots, g^{k^{2n}}\}$ where $k \overset{\$}{\leftarrow} \mathbb{Z}_p^*$, compute $g^{k^{n+1}}$. The n-DHE assumption states that, for any adversary $\mathcal{A}$, it holds that

$$\Pr\left[\mathcal{A}(g, g^k, g^{k^2}, \ldots, g^{k^n}, g^{k^{n+2}}, \ldots, g^{k^{2n}}) = g^{k^{n+1}}\right] = \mathsf{negl}(s).$$

Simply stated, even given the information in $\mathcal{V}$, the (polynomially bounded) adversary is unable to solve the problem with any nonnegligible probability.

One-time pad and nonces. One-time pad is a method of encrypting data that exhibits perfect secrecy [Katz and Lindell 2007] if implemented correctly. In one-time pad encryption, a message M is encrypted using a random key K that: (i) has (at least) the same size as M, and (ii) is used exactly once. The encryption is performed via a XOR operation as $M \oplus K$. In our work, we use an alternative form of one-time pad that uses modular arithmetic. In particular, we encrypt a number $M \in \mathbb{Z}_p$ by a random (used once) key $K \in \mathbb{Z}_p^*$ as $(M+K) \mod p$. While slightly less efficient than a XOR operation, this alternative mode of one-time pad offers the same security as the original, and will be particularly helpful in our proposed techniques. Finally, we refer to any key that is used just once as a *nonce*.

2.2. Related Work

The closest schemes to ours are PIRS [Yi et al. 2009] and DiSH [Nath and Venkatesan 2013], which both focus on authenticating results for `group-by,sum` queries. In these works, the stream consists of unaggregated tuples. The server's task is to perform a `group-by` operation to collate the tuples into predefined groups, and then to compute an aggregate such as `sum` on each group. In both PIRS and DiSH the owner maintains a small summary on the observed stream, which facilitates verifying the result correctness. PIRS is a probabilistic protocol where the client is the owner itself. Due to its simplified model and relaxed security guarantees, PIRS is lightweight. On the other hand, DiSH is a cryptographic technique which assumes that the clients are parties untrusted by the owner. The clients can directly communicate, though, with the owner to receive the summary. In order to address the challenge that clients cannot possess any secret material from the owner, DiSH employs expensive cryptographic primitives such as modular exponentiations during authentication and verification.

Note that PIRS and DiSH do not directly capture our general architecture (described in Section 1), where the owner and the clients are different physical entities that communicate with each other via the untrusted server. In order to adapt PIRS and DiSH to our scenario, the owner must rely on some other message authentication technique for securely forwarding the summary to the clients via the server, for instance, using HMACs and public-key digital signatures [Menezes et al. 1996], respectively. This inflicts extra overhead on both the owner (for authentication) and the client (for verification).

Also related to our work is the use of message authentication codes (MACs) that are homomorphic, since they allow the linear combination of messages from different sources, along with the corresponding combination of the MACs of these messages. In our schemes, we also need to utilize the signatures of multiple sources (i.e., owner machines), but these signatures must be properly computed and combined in order to authenticate the result of different operators. Homomorphic MACs were first proposed in Agrawal and Boneh [2009] for network coding applications and have since been widely used, such as in Boneh and Freeman [2011] for evaluating multivariate polynomials on signed data, or in Papadopoulos et al. [2011] for computing simple statistics in sensor networks. We emphasize that homomorphism is a property that message authentication techniques may exhibit, but not a tool for automatically authenticating general operators/functions (such as the ones examined in this article) over distributed data. In particular, we are not aware of any prior work that has addressed the general linear queries such as matrix multiplication and dot products that we study here.

Authentication results have also been shown for other problems and models. In the context of outsourced databases, there are techniques that address snapshot relational queries, such as ranges and joins [Devanbu et al. 2003; Pang and Tan 2004; Pang et al. 2005; Li et al. 2006; Yang et al. 2009], as well as continuous range queries [Li et al. 2007; Papadopoulos et al. 2007]. All these methods rely on authenticated data structures (such as Merkle trees) that are maintained by the owner and signed with public-key cryptosystems. These data structures are large: linear in the size of the input data. There has also been a line of work on verifying simple aggregate computations in distributed networks, such as `sum`, `min/max`, and `count` [Garofalakis et al. 2007; Papadopoulos et al. 2011; Nath et al. 2009]. In this setting, the machines are organized into a tree hierarchy. The internal (potentially untrusted) nodes perform in-network aggregation as they route information from the leaves to the root (sink).

Our work differs from all these prior efforts in several important respects. First, we consider fundamental problems that can be adapted to solve a wide range of important queries in stream outsourcing. Second, our constructions impose a very low overhead to all parties. In particular, they do not entail the costly exponentiation operations involved in DiSH, and do not require the owner to maintain sophisticated structures, as in the database outsourcing solutions. Lastly, unlike PIRS, our work comes with strong cryptographic guarantees that formally demonstrate the security and robustness of our schemes against malicious activity and errors.

Some related studies have been conducted within the theory community. The model of *annotated streams* allows the server to insert some “advice” into a stream to help a client compute a function of interest. This model was applied to problems such as recovering information about particular items from the stream, functions of the item frequencies (such as the frequency moments), and some graph computations [Chakrabarti et al. 2009]. The costs of these protocols are typically sublinear but polynomial in the size of the stream. These costs were subsequently reduced to logarithmic for some key problems, but only when there are multiple rounds of communication between the data owner and server [Cormode et al. 2011]. General computations can be authenticated with a pass over the data, but this can require thousands of rounds of interaction between the parties [Goldwasser et al. 2008].

The topic has also attracted interest in the study of cryptography, with similar motivations to those we present here. Gennaro et al. [2010] outline a general scheme for verification of computation modeled as a Boolean circuit. The technique involves evaluating Yao’s garbled circuit construction within fully homomorphic encryption—an approach that has not yet reached close to an efficient implementation. The related problem of memory delegation [Chung et al. 2011] is to allow one to delegate the storage of a stream of data to a third party, and then retrieve pieces of the data or computations thereon. Results in this area also require the use of fully homomorphic encryption. Of more practical relevance is the work of Papamanthou et al. [2011] that considers the verification of set operations such as intersection, union, and set difference. The implementation is based on evaluations of encryptions of polynomials encoding the members of the set; the set operations considered are complementary to the linear algebraic computations we focus on here. Also related to our work is that of Cormode et al. [2012]. The central result in this work is an implementation of Goldwasser et al. [2008], engineered for practicality. A limitation is that it still requires many rounds of interaction. Moreover, this interaction reveals secrets held by the data owner, limiting the use of the protocol for multiple iterations. In contrast, our results do not reveal anything about their secret keys and so can be iterated over many timestamps. Noninteractive results are shown that do not reveal key material, but these involve exponentially more communication costs to the data owner.

In the conference version of this article [Papadopoulos et al. 2013], we presented the initial framework for authenticating linear algebraic queries on data streams. In this manuscript we extend the work of Papadopoulos et al. [2013] in a number of directions.

First, we improve the verification cost of our basic constructions using a novel nonce chaining technique (Section 5). We then extend our basic methods to the case of sliding window queries (Section 6). Next, we present new extensions of our techniques for handling concurrent subset queries by multiple clients (Section 7). An important final addition is the detailed presentation in this article of how join queries can be handled and proven secure in our framework. The case of join queries was only briefly sketched in Papadopoulos et al. [2013].

3. FORMULATION

Section 3.1 defines the system setting outlined in Section 1 as a formal protocol executed by the involved parties. Section 3.2 presents the security model.

3.1. Stream Authentication Protocol

The definition that follows formulates a stream authentication scheme, assuming a security parameter s.

Definition 3.1. A stream authentication scheme is a set of five algorithms (KeyGen, Update, Sign, Combine, Verify) running in time polynomial in s and described as follows.

$(sk, pub) \leftarrow \mathsf{KeyGen}(1^s)$ is a probabilistic algorithm that takes as input a security parameter s, and outputs secret key sk and public information pub.

$\mathcal{S}_i \leftarrow \mathsf{Update}(i, sk, \mathcal{S}_i, t)$ is a (potentially) probabilistic algorithm that takes as input id i, secret key sk, summary $\mathcal{S}_i$, and incoming tuple t. It produces updated summary $\mathcal{S}_i$.

$\sigma_{i,\tau} \leftarrow \mathsf{Sign}(i, sk, \mathcal{S}_i, \tau)$ is a (potentially) probabilistic algorithm that takes as input id i, secret key sk, summary $\mathcal{S}_i$, and epoch τ. It produces signature $\sigma_{i,\tau}$.

$\pi_\tau := \mathsf{Combine}(U_\sigma, U_\mathcal{X}, pub)$ is a deterministic algorithm that takes as input a set U_σ of signatures and a set $U_\mathcal{X}$ of streams, along with public info pub. It produces proof π_τ.

Yes|**No** $:= \mathsf{Verify}(sk, \pi_\tau, res_\tau, \tau)$ is a deterministic algorithm that takes as input secret key sk, proof π_τ, result res_τ, and epoch τ. It outputs a string that is either **Yes** or **No**.

The protocol is executed in the following stages.

—*Setup.* The protocol commences with an offline setup phase. The owner runs KeyGen and produces a secret key sk and public info pub. It installs a unique identifier i, key sk, and an initial summary $\mathcal{S}_i$ in every machine M_i, and sends pub to the server. It also securely provides the client with sk, for example, via an SSL channel. Next, it concludes the setup phase and sets the system into motion.

—*Update and signing at M_i.* Whenever a new tuple t is generated by M_i, the machine runs Update before forwarding t to the server. This algorithm uses key sk and t on current summary $\mathcal{S}_i$ and outputs a new summary that substitutes for the old one. At the end of epoch τ, M_i runs Sign on i, sk, τ, and current summary $\mathcal{S}_i$ to produce a signature $\sigma_{i,\tau}$, that is sent to the server.

—*Result and proof generation at the server.* At the end of epoch τ the server receives new signatures from the machines. It computes and sends result res_τ to the client in response to continuous query Q. Moreover, it transmits a proof π_τ that is produced by algorithm Combine on U_σ, $U_\mathcal{X}$, and pub.

—*Verification at the client.* At the end of epoch τ the client receives from the server a new result res_τ, accompanied by a new proof π_τ. It verifies result correctness via Verify, which combines res_τ with π_τ and the owner's secret key sk. The output is **Yes** if verification succeeds, and **No** otherwise. Note that the client is *stateless*, that is, it verifies with respect to the entire history of the data streams, not since the last successful verification.

The next definition formulates scheme correctness.

Definition 3.2. A stream authentication scheme is *correct* if the following condition holds. For any security parameter s, let (sk, pub) be any output of algorithm $\mathsf{KeyGen}(1^s)$. Let $\mathcal{X}_i(\tau)$ be any stream observed by M_i up until τ, and $Q(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\})$ the result of query Q at τ. Let $\mathcal{S}_i$ be the summary computed by executing Update on sk and on every $t \in \mathcal{X}_i(\tau)$. Let $\sigma_{i,\tau}$ be the signature produced by M_i via $\mathsf{Sign}(i, sk, \mathcal{S}_i, \tau)$. Finally, let π_τ be the proof that is output by $\mathsf{Combine}(U_\sigma, U_\mathcal{X}, pub)$ for some query-specific $U_\sigma, U_\mathcal{X}$. Then, $\mathsf{Verify}(sk, \pi_\tau, res_\tau, \tau)$ returns **Yes** when

$$res_\tau = Q\left(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\}\right).$$

This correctness definition also entails that the result is *complete*, that is, all the tuples seen by the data owner must be processed to produce the desired result. Note that scheme correctness does not specify the output of Verify in case $res_\tau \neq Q(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\})$. This is captured by the definition of security, included in the next section.

3.2. Security Definition

The adversary $\mathcal{A}$ may be the server, or any other entity other than the owner's machines and the client. $\mathcal{A}$ is allowed to access the raw data streams, that is, data privacy is out of the scope of our work. Nevertheless, $\mathcal{A}$ may tamper with the outputs at any epoch. Our security goal against $\mathcal{A}$ is result correctness that jointly guarantees: (i) *integrity* (i.e., that the result is not falsified) and (ii) *freshness* (i.e., that the result is up-to-date).

We rigorously model security via the following experiment, a variation of the standard existential unforgeability under an adaptive chosen message attack [Katz and Lindell 2007].

Experiment $\mathsf{Exp}_\mathcal{A}(1^s)$

(1) Pair (sk, pub) is output by KeyGen, and pub is given to $\mathcal{A}$.
(2) $\mathcal{A}$ is given oracle access to Sign as follows: $\mathcal{A}$ presents a triplet (T, i, τ'), where T is a set of tuples. The oracle keeps record of all submitted queries, and rejects a query that requests a signature for a certain (i, τ') more than once. If it does not reject, the oracle initializes $\mathcal{S}_i = 0$ and runs $\mathsf{Update}(i, sk, \mathcal{S}_i, t)$ for every $t \in T$, producing summary $\mathcal{S}_i$. It then runs $\mathsf{Sign}(i, sk, \mathcal{S}_i, \tau')$, and returns the result to $\mathcal{A}$.
(3) $\mathcal{A}$ outputs a pair (res^*_τ, π^*_τ), with the restriction that:
—$res^*_\tau \neq Q(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\})$;
—Sign was not queried for any triplet (T, i, τ'), such that $(\tau' = \tau) \wedge (T \neq \mathcal{X}_i(\tau))$.
(4) If $\mathsf{Verify}(sk, \pi^*_\tau, res^*_\tau, \tau)$ returns **Yes**, then output 1; otherwise output 0.

We say that a stream authentication scheme is *secure* if no adversary $\mathcal{A}$ can succeed in the preceding experiment with nonnegligible probability, that is, if it holds that

$$\Pr[\mathsf{Exp}_\mathcal{A}(1^s) = 1] = \mathsf{negl}(s),$$

where the probability is taken over the random choice of sk and the random coin tosses of $\mathcal{A}$.

Simply stated, during the attack $\mathcal{A}$ is allowed to obtain (through the oracle) any number of signatures for any machine and stream of its choice, at any epoch other than the epoch τ for which it launches the attack. At τ, $\mathcal{A}$ is only allowed access to the valid signatures produced by the machines. $\mathcal{A}$ then launches the attack by presenting a pair (res^*_τ, π^*_τ) such that res^*_τ is different from the actual result. Our aim is to provide protocols that are secure against such attacks and will not accept any such incorrect results.

4. BASIC CONSTRUCTIONS

In this section we present constructions that can be used as building blocks for designing authentication schemes for a wide range of query types. In particular, we design techniques for authenticating dynamic vector sums (Section 4.1), dynamic matrix products (Section 4.2), and dynamic dot products (Section 4.3). Throughout, we consider a security parameter s, a prime p whose bit size is $\Theta(s)$, and a PRF $F : \mathbb{Z}_p \times \{0, 1\}^* \rightarrow \mathbb{Z}_p$, which are all known as *globals* to all parties. We assume that all the stream values and aggregate results belong to $\mathbb{Z}_p$. This is without loss of generality, since: (i) for practical values of s, $\mathbb{Z}_p$ is large enough for any application, and (ii) application domains that involve negative integers work directly for p large enough, while those that involve real numbers can be converted to $\mathbb{Z}_p$ via scaling and rounding.

4.1. Dynamic Vector Sum Authentication

We focus on m machines M_i, and consider a vector $\mathbf{a}_i$ with n entries that is dynamically updated as new tuples t are generated by M_i. Each tuple $t \in \mathcal{X}_i$ is of the form (j, v), and updates $\mathbf{a}_i$ by adding v to $\mathbf{a}_i[j]$. The client's query Q requests the sum of the vectors produced by all machines at every epoch τ, that is,

$$Q\left(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\}\right) = \sum_{i=1}^{m}\mathbf{a}_i = \left[\sum_{i=1}^{m}\mathbf{a}_i[1], \ldots, \sum_{i=1}^{m}\mathbf{a}_i[n]\right].$$

We term such a query a *dynamic vector sum* query and present next a scheme called DVS for authenticating it.

Figure 2 presents the DVS construction that instantiates the general stream authentication protocol outlined in Section 3.1. The intuition behind this construction is straightforward: the summary $\mathcal{S}_i$ captures the current state of vector $\mathbf{a}_i$, in such a way that the adversary, lacking knowledge of the secret sk, has no way of finding another vector $\mathbf{a}^*_i$ that would have the same summary, even given access to other signatures. The signature $\sigma_{i,\tau}$ includes additional information (the nonce $r_{i,\tau}$) that prevents the server from reusing the same signature at different epochs or for different machines. All operations are performed modulo p (i.e., in $\mathbb{Z}_p$).

Every summary is initialized to 0 during the setup phase. Algorithm Update works in a way such that $\mathcal{S}_i$ is equal to the dot product $\mathbf{k} \cdot \mathbf{a}_i$, where $\mathbf{k} = [k_1, \ldots, k_n]$. Sign injects a machine- and time-dependent key $r_{i,\tau}$ used once. Observe that every k_j and $r_{i,\tau}$ value is produced with sk via PRF F, where "element", "machine", and "epoch" are string labels. Combine simply adds all the signatures retrieved from the machines. Combine does not need any public information from the owner and thus pub is set to a null value in KeyGen. The client assumes all m machines are involved in the protocol when executing Verify. In general, the client must know exactly which machines participate in the protocol in order to properly calculate the $r_{i,\tau}$ values. As an additional remark, observe that DVS can be used even when only a single machine is involved. In this case, DVS essentially supports dynamic vector authentication.

Correctness and security. The following theorem proves the correctness of DVS.

THEOREM 4.1. DVS *is correct.*

```
KeyGen($1^s$)
1. $k \xleftarrow{\$} \mathbb{Z}_p^*$
2. Output $sk = k$ and $pub = \perp$

Update($i, sk, \mathcal{S}_i, t$)
1. Parse $t$ as $(j, v)$, and $sk$ as $k$
2. $k_j = F_k(\text{"element"}\|j)$
3. $\mathcal{S}_i = \mathcal{S}_i + k_j \cdot v$
4. Output $\mathcal{S}_i$

Sign($i, sk, \mathcal{S}_i, \tau$)
1. $r_{i,\tau} = F_k(\text{"machine"}\|i\|\text{"epoch"}\|\tau)$
2. $\sigma_{i,\tau} = \mathcal{S}_i + r_{i,\tau}$
3. Output $\sigma_{i,\tau}$

Combine($\bigcup_{i=1}^{m}\{\sigma_{i,\tau}\}, \emptyset, pub$)
1. Output $\pi_\tau = \sum_{i=1}^{m} \sigma_{i,\tau}$

Verify($sk, \pi_\tau, res_\tau, \tau$)
1. Parse $sk = k$ and $res_\tau$ as a $n$-element vector
2. For $i = 1$ to $m$, $r_{i,\tau} = F_k(\text{"machine"}\|i\|\text{"epoch"}\|\tau)$
3. Initialize $\pi = \sum_{i=1}^{m} r_{i,\tau}$
4. For $j = 1$ to $n$
5.     $k_j = F_k(\text{"element"}\|j)$
6.     $\pi = \pi + k_j \cdot res_\tau[j]$
7. If $\pi = \pi_\tau$ output Yes, otherwise No
```

Fig. 2. The DVS construction.

Proof. Let the actual result of Q at τ be $Q(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\}) = \sum_{i=1}^{m} \mathbf{a}_i$, where $\mathbf{a}_i[j] = \sum_{t\in\mathcal{X}_i(\tau)\wedge t.j=j} t.v$. Observe that, after executing Update for all $t \in \mathcal{X}_i(\tau)$ at any M_i, $\mathcal{S}_i = \sum_{j=1}^{n} k_j \cdot \mathbf{a}_i[j]$. Then, Combine calculates $\pi_\tau = (\sum_{j=1}^{n} k_j \cdot (\sum_{i=1}^{m} \mathbf{a}_i[j])) + \sum_{i=1}^{m} r_{i,\tau}$. Now notice that, if res_τ passed in Verify is equal to $Q(\bigcup_{i=1}^{m}\{\mathcal{X}_i(\tau)\})$, then the algorithm computes a π that is equal to the π_τ calculated earlier, hence the output is **Yes**. □

We next state the security of DVS (the proof is in online Appendix A.1 accessible in the ACM Digital Library).

Theorem 4.2. *If F is a PRF, then* DVS *is secure.*

Performance. Every machine M_i needs to store only the key sk, and its id i. Therefore, the memory consumption is $O(s + \log m)$, where s is the security parameter that dictates the size of sk, and $\log m$ is the size of the machine id (where m is the number of machines). Since the size of p is $\Theta(s)$, the communication cost between any two parties is $O(s)$. For any practical application, s and $\log m$ can be regarded as constants that do not exceed 20 bytes. Note that we implement F as an HMAC [Menezes et al. 1996], which involves two hash operations. Both Update and Sign entail a constant number of modular multiplications/additions and hashes. The overhead for the server is $O(m)$ modular additions. Finally, the burden at the client is $O(m + n)$ modular additions/multiplications and hashes.

4.2. Dynamic Matrix Product Authentication

We focus on two machines, M_a and M_b. We consider an $n_a \times n$ matrix $\mathbf{A}$ and an $n \times n_b$ matrix $\mathbf{B}$. Matrix $\mathbf{A}$ (respectively, $\mathbf{B}$) is dynamically updated as new tuples are generated by M_a (M_b). Each tuple $t \in \mathcal{X}_a$ (respectively, $t \in \mathcal{X}_b$) is of the form (i, j, v) and updates $\mathbf{A}$ ($\mathbf{B}$) by adding v to $\mathbf{A}[i][j]$ ($\mathbf{B}[i][j]$). The client's query Q requests the matrix

```
KeyGen($1^s$)
1. $k \xleftarrow{\$} \mathbb{Z}_p^*$
2. Output $sk = k$ and $pub = \perp$

Update($a, sk, \mathcal{S}_a, t$)
1. Parse $t$ as $(i, j, v)$, and $sk$ as $k$
2. $k_{a,i} = F_k(\text{“machine”}\|a\|\text{“element”}\|i)$
3. $\mathcal{S}_a[j] = \mathcal{S}_a[j] + k_{a,i} \cdot v$
4. Output $\mathcal{S}_a$

Sign($a, sk, \mathcal{S}_a, \tau$)
1. For $j = 1$ to $n$
2.     $r_{a,\tau}[j] = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau\|\text{“}r\text{”}\|j)$
3.     $\rho_{a,\tau}[j] = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau\|\text{“}\rho\text{”}\|j)$
4.     $r_{b,\tau}[j] = F_k(\text{“machine”}\|b\|\text{“epoch”}\|\tau\|\text{“}r\text{”}\|j)$
5.     $\sigma_{a,\tau}[j] = [(\mathcal{S}_a[j] + r_{a,\tau}[j]), (\mathcal{S}_a[j] \cdot r_{b,\tau}[j] + \rho_{a,\tau}[j])]$
6. Output $\sigma_{a,\tau}$

Combine($\{\sigma_{a,\tau}, \sigma_{b,\tau}\}, \emptyset, pub$)
1. $\pi_\tau = \sum_{j=1}^{n}(\sigma_{a,\tau}[j][1] \cdot \sigma_{b,\tau}[j][1] - \sigma_{a,\tau}[j][2] - \sigma_{b,\tau}[j][2])$
2. Output $\pi_\tau$

Verify($sk, \pi_\tau, res_\tau, \tau$)
1. Parse $sk$ as $k$ and $res_\tau$ as a $n_a \times n_b$ matrix
2. For $j = 1$ to $n$
3.     $r_{a,\tau}[j] = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau\|\text{“}r\text{”}\|j)$
4.     $\rho_{a,\tau}[j] = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau\|\text{“}\rho\text{”}\|j)$
5.     $r_{b,\tau}[j] = F_k(\text{“machine”}\|b\|\text{“epoch”}\|\tau\|\text{“}r\text{”}\|j)$
6.     $\rho_{b,\tau}[j] = F_k(\text{“machine”}\|b\|\text{“epoch”}\|\tau\|\text{“}\rho\text{”}\|j)$
7. Initialize $\pi = \sum_{j=1}^{n}(r_{a,\tau}[j] \cdot r_{b,\tau}[j] - \rho_{a,\tau}[j] - \rho_{b,\tau}[j])$
8. For $i = 1$ to $n_a$, $k_{a,i} = F_k(\text{“machine”}\|a\|\text{“element”}\|i)$
9. For $j = 1$ to $n_b$, $k_{b,j} = F_k(\text{“machine”}\|b\|\text{“element”}\|j)$
10. $\pi = \pi + \sum_{i \in [n_a], j \in [n_b]} k_{a,i} \cdot k_{b,j} \cdot res_\tau[i][j]$
11. If $\pi = \pi_\tau$ output **Yes**, otherwise **No**
```

Fig. 3. The DMP construction.

product, denoted by **AB**, between **A** and **B** at every epoch τ. We term such a query as a dynamic matrix product query. We next present a scheme, called DMP, for dynamic matrix product query authentication.

Figure 3 presents the DMP construction. The technique takes advantage of the following property of matrix multiplication. Let $\mathbf{A} = [\mathbf{a}_1\mathbf{a}_2 \ldots \mathbf{a}_n]$, where $\mathbf{a}_j$ denotes the jth column of **A**. Also let $\mathbf{B} = [\mathbf{b}_1\mathbf{b}_2 \ldots \mathbf{b}_n]^{\mathrm{T}}$, where $\mathbf{b}_j$ is the jth row of **B**. Then it holds that

$$Q(\mathcal{X}_a(\tau) \cup \mathcal{X}_b(\tau)) = \mathbf{AB} = \sum_{j=1}^{n} \mathbf{a}_j \otimes \mathbf{b}_j,$$

where $\mathbf{a}_j \otimes \mathbf{b}_j$ is the outer product of vectors $\mathbf{a}_j$, $\mathbf{b}_j$, such that

$$\mathbf{a}_j \otimes \mathbf{b}_j = \begin{bmatrix} \mathbf{a}_j[1]\mathbf{b}_j[1] & \mathbf{a}_j[1]\mathbf{b}_j[2] & \ldots & \mathbf{a}_j[1]\mathbf{b}_j[n_b] \\ \mathbf{a}_j[2]\mathbf{b}_j[1] & \mathbf{a}_j[2]\mathbf{b}_j[2] & \ldots & \mathbf{a}_j[2]\mathbf{b}_j[n_b] \\ \ldots & \ldots & \ldots & \ldots \\ \mathbf{a}_j[n_a]\mathbf{b}_j[1] & \mathbf{a}_j[n_a]\mathbf{b}_j[2] & \ldots & \mathbf{a}_j[n_a]\mathbf{b}_j[n_b] \end{bmatrix}.$$

M_a (respectively, M_b) can create a summary $\mathcal{S}_a[j]$ ($\mathcal{S}_b[j]$) for vector $\mathbf{a}_j$ ($\mathbf{b}_j$) in a similar manner to DVS. We can then compute a summary of $\mathbf{a}_j \otimes \mathbf{b}_j$ from the product $\mathcal{S}_a[j] \cdot \mathcal{S}_b[j]$:

for each entry of this outer product, there is a corresponding term in $\mathcal{S}_a[j] \cdot \mathcal{S}_b[j]$, scaled by a secret value (i.e., the product of the two corresponding keys). In other words, we obtain a summary of the outer product result matrix with similar properties to the DVS summary for a single vector. Since matrix multiplication can be expressed as a sum of outer products, we can use n different summary products $\mathcal{S}_a[j] \cdot \mathcal{S}_b[j]$ (i.e., one for each outer product $\mathbf{a}_j \otimes \mathbf{b}_j$) and build a summary for matrix product $\mathbf{AB}$ by summing them up.

We assume M_a knows that M_b participates in the query and vice versa (this information is part of the query description). The summaries $\mathcal{S}_a, \mathcal{S}_b$ are both initialized to zero n-element vectors during the setup phase. Algorithms Update and Sign are presented in the context of M_a. Now $\mathcal{S}_a$ contains n entries, one for each column. The case of M_b is symmetric; $\mathcal{S}_b$ also includes n entries, but one for each row. This can be achieved by instead parsing t as (j, i, v) in line 1 of Update, and proceeding accordingly.

To provide security for these summaries, the Sign function produces composite signatures $\sigma_{a,\tau}[j], \sigma_{b,\tau}[j]$, each consisting of two elements/signatures. In particular, their first elements ($\sigma_{a,\tau}[j][1]$ and $\sigma_{b,\tau}[j][1]$) integrate machine-, time-, and column-/row-dependent values r to mask the summaries as in DVS. In order to produce a proof for summaries of the form $\mathcal{S}_a[j] \cdot \mathcal{S}_b[j]$, the server needs to multiply $\sigma_{a,\tau}[j][1]$ with $\sigma_{b,\tau}[j][1]$. However, observe that terms $r_{a,\tau}[j] \cdot \mathcal{S}_b[j]$ and $r_{b,\tau}[j] \cdot \mathcal{S}_a[j]$ will appear in the resulting proof, which are hard to verify by the client without $\mathcal{S}_a[j]$ and $\mathcal{S}_b[j]$. Hence, the machines provide extra information (namely signatures $\sigma_{a,\tau}[j][2], \sigma_{b,\tau}[j][2]$) that enable the server to remove these values from the proof. To ensure security, these signatures incorporate new one-time keys (denoted as ρ).

Based on the preceding, Combine now takes a combination of $2n$ elements together to build a compact proof that includes the summary of the whole product matrix. Note that π_τ is just a single value modulo p. Similar to DVS, Combine does not need any public information from the owner and thus *pub* is set to a null value in KeyGen. Finally, algorithm Verify needs to include the various masking values created by M_a and M_b for each of their n parallel summaries and outputs **Yes** only if the proof computed for the claimed result matches the provided proof π_τ.

Correctness and security. The following two theorems state the correctness and security of DMP.

THEOREM 4.3. DMP *is correct.*

PROOF. Let the actual result of Q at τ be $Q(\{\mathcal{X}_a(\tau) \cup \mathcal{X}_b(\tau)\}) = \mathbf{AB}$, where $\mathbf{A}[i][j] = \sum_{t \in \mathcal{X}_a(\tau) \wedge (t.i=i) \wedge (t.j=j)} t.v$ and $\mathbf{B}[i][j] = \sum_{t \in \mathcal{X}_b(\tau) \wedge (t.i=i) \wedge (t.j=j)} t.v$. Observe that, after executing Update for all $t \in \mathcal{X}_a(\tau)$ and $t \in \mathcal{X}_b(\tau)$ at M_a and M_b, respectively, $\mathcal{S}_a[j] = \sum_{i=1}^{n_a} k_{a,i} \cdot \mathbf{a}_j[i]$ and $\mathcal{S}_b[j] = \sum_{i=1}^{n_b} k_{b,i} \cdot \mathbf{b}_j[i]$. Moreover, notice that

$$\pi_\tau = \sum_{j=1}^{n} (\mathcal{S}_a[j] \cdot \mathcal{S}_b[j] + r_{a,\tau}[j] \cdot r_{b,\tau}[j] - \rho_{a,\tau}[j] - \rho_{b,\tau}[j]).$$

However, it holds that

$$\begin{aligned}\sum_{j=1}^{n} (\mathcal{S}_a[j] \cdot \mathcal{S}_b[j]) &= \sum_{j=1}^{n} \left(\sum_{i=1}^{n_a} k_{a,i} \cdot \mathbf{a}_j[i] \cdot \sum_{i=1}^{n_b} k_{b,i} \cdot \mathbf{b}_j[i] \right) \\ &= \sum_{i \in [n_a], j \in [n_b]} k_{a,i} \cdot k_{b,j} \cdot \sum_{z=1}^{n} \mathbf{a}_z[i] \cdot \mathbf{b}_z[j] \\ &= \sum_{i \in [n_a], j \in [n_b]} k_{a,i} \cdot k_{b,j} \cdot \mathbf{AB}[i][j].\end{aligned}$$

If res_τ is equal to $\mathbf{AB}$, then it is easy to see that the π computed in Verify is equal to π_τ and thus the algorithm outputs **Yes**. This concludes our proof. □

Theorem 4.4. *If F is a PRF, then* DMP *is secure.*

For the proof, see Appendix A.2.

Performance. The memory consumption and computational cost of Update at each machine is the same as in DVS. Due to the n masked summaries, algorithm Sign involves $O(n)$ modular additions/multiplications and hashes, whereas the communication cost between a machine and the server becomes $O(n)$. The server computes $O(n)$ modular additions/multiplications in Combine. Finally, the client receives a constant-sized proof, but Verify entails $O(n_a + n_b + n)$ hashes and $O(n_a n_b)$ modular additions/multiplications, proportional to the cost of reading the result. This is reduced if the result matrix is sparse: then, the time taken is proportional to the number of nonzero entries, which can be much lower.

Note that this protocol substantially reduces the burden on the data owner compared to the cost it would pay to perform the matrix multiplication itself. Without outsourcing, the data owner would have to store the $O(n^2)$ entries of the matrices and perform a superquadratic amount of work to carry out the multiplication. Here, the data owner's requirements are reduced to $O(n)$ storage per machine and constant work per update.

4.3. Dynamic Dot Product Authentication

We focus on two machines, M_a and M_b, and consider n-element vectors $\mathbf{a}, \mathbf{b}$. Vector $\mathbf{a}$ (respectively, $\mathbf{b}$) is dynamically updated as new tuples are generated by M_a (M_b). Each tuple $t \in \mathcal{X}_a$ (respectively, $t \in \mathcal{X}_b$) is of the form (j, v) and updates $\mathbf{a}$ ($\mathbf{b}$) by adding v to $\mathbf{a}[j]$ ($\mathbf{b}[j]$). The client's query Q requests the dot product between $\mathbf{a}$ and $\mathbf{b}$ at every epoch τ, that is,

$$Q(\mathcal{X}_a(\tau) \cup \mathcal{X}_b(\tau)) = \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} \mathbf{a}[i] \cdot \mathbf{b}[i].$$

We refer to such a query as a *dynamic dot product* query and present a scheme called DDP for authenticating it.

Figure 4 presents the DDP construction. Similar to DMP, we assume M_a knows that M_b participates in the query and vice versa. Algorithms Update and Sign are described in the context of M_a. The case of M_b is symmetric, with the vital difference that the summary is updated as $\mathcal{S}_b = \mathcal{S}_b + k^{n-j+1} \cdot v$ in line 2 of Update. The summaries $\mathcal{S}_a, \mathcal{S}_b$ are initialized to 0 during the setup phase. We make use of a (multiplicative) cyclic group $\mathbb{G}$ of order p with generator g, whose specifications are public and where the n-DHE problem is hard (see Section 2.1).

Note that the dot product of two vectors is the trace of their outer product. We use this fact to construct the protocol. We derive a signature of the outer product $\mathbf{a} \otimes \mathbf{b}$ in a similar manner to DMP, where each element of the resulting matrix is scaled with a secret key. Furthermore, certain machine- and time-dependent masking is performed via the r and ρ values. The server is then responsible for removing certain elements $\mathbf{a}[i] \cdot \mathbf{b}[j]$, which are scaled by $k^{i+(n-j+1)}$, from the signature in Combine. Specifically, the server does this for every $i \neq j$ (i.e., all elements but those in the diagonal).

In order to facilitate this task, the owner provides some public information *pub* to the server concerning the scalar values $k^{i+(n-j+1)}$, with the exception of k^{n+1}. These keys are given as exponents of generator $g \in \mathbb{G}$. This is necessary because, otherwise, the server could trivially retrieve k^{n+1} as $k^{n+i+1} \cdot (k^i)^{-1} \mod p$ for some i, where $(k^i)^{-1}$ is the multiplicative inverse of k^i modulo p. This cannot happen if the keys are in the

```
KeyGen($1^s$)
1. $k \xleftarrow{\$} \mathbb{Z}_p^*$
2. $pub = \{g^{k^j}\}_{j \in [2n] \setminus \{n+1\}}$
3. Output $sk = k$ and $pub$

Update($a, sk, \mathcal{S}_a, t$)
1. Parse $t$ as $(j, v)$, and $sk$ as $k$
2. $\mathcal{S}_a = \mathcal{S}_a + k^j \cdot v$
3. Output $\mathcal{S}_a$

Sign($a, sk, \mathcal{S}_a, \tau$)
1. $r_{a,\tau} = F_k(\text{“machine”} \| a \| \text{“epoch”} \| \tau \| \text{“}r\text{”})$
2. $\rho_{a,\tau} = F_k(\text{“machine”} \| a \| \text{“epoch”} \| \tau \| \text{“}\rho\text{”})$
3. $r_{b,\tau} = F_k(\text{“machine”} \| b \| \text{“epoch”} \| \tau \| \text{“}r\text{”})$
4. $\sigma_{a,\tau} = [(\mathcal{S}_a + r_{a,\tau}), (\mathcal{S}_a \cdot r_{b,\tau} + \rho_{a,\tau})]$
5. Output $\sigma_{a,\tau}$

Combine($\{\sigma_{a,\tau}, \sigma_{b,\tau}\}, \{\mathcal{X}_a(\tau), \mathcal{X}_b(\tau)\}, pub$)
1. Parse $pub$ as $\{g^{k^i}\}_{i \in [2n] \setminus \{n+1\}}$
2. Compute $\mathbf{a}$ and $\mathbf{b}$ from $\mathcal{X}_a(\tau)$ and $\mathcal{X}_b(\tau)$, respectively
3. Compute $\mathbf{c} = \mathbf{a} \otimes \mathbf{b}$
4. $\pi_\tau = g^{(\sigma_{a,\tau}[1] \cdot \sigma_{b,\tau}[1] - \sigma_{a,\tau}[2] - \sigma_{b,\tau}[2])}$
5. $\pi_\tau = \pi_\tau \cdot \left[\prod_{i,j \in [n] \wedge i \neq j} \left(g^{k^{i+(n-j+1)}}\right)^{\mathbf{c}[i][j]}\right]^{-1}$
6. Output $\pi_\tau$

Verify($sk, \pi_\tau, res_\tau, \tau$)
1. Parse $sk$ as $k$ and $res_\tau$ as a value in $\mathbb{Z}_p$
2. $r_{a,\tau} = F_k(\text{“machine”} \| a \| \text{“epoch”} \| \tau \| \text{“}r\text{”})$
3. $\rho_{a,\tau} = F_k(\text{“machine”} \| a \| \text{“epoch”} \| \tau \| \text{“}\rho\text{”})$
4. $r_{b,\tau} = F_k(\text{“machine”} \| b \| \text{“epoch”} \| \tau \| \text{“}r\text{”})$
5. $\rho_{b,\tau} = F_k(\text{“machine”} \| b \| \text{“epoch”} \| \tau \| \text{“}\rho\text{”})$
6. Initialize $\pi = g^{(r_{a,\tau} \cdot r_{b,\tau} - \rho_{a,\tau} - \rho_{b,\tau})}$
7. $\pi = \pi \cdot g^{(k^{n+1} \cdot res_\tau)}$
8. If $\pi = \pi_\tau$ output **Yes**, otherwise **No**
```

Fig. 4. The DDP construction.

exponent of g due to the n-DHE assumption (we will use this fact later in our rigorous proof). All computations in Verify are performed in the exponent of g. Following this, the output π_τ should contain solely the contribution from elements on the diagonal of the outer product, all scaled by k^{n+1}, plus the masking values.

Correctness and security. The following theorems state the correctness and security of DDP, respectively.

THEOREM 4.5. DDP *is correct.*

PROOF. Let the actual result of Q at τ be $Q(\{\mathcal{X}_a(\tau) \cup \mathcal{X}_b(\tau)\}) = \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^n \mathbf{a}[i] \cdot \mathbf{b}[i]$, where $\mathbf{a}[j] = \sum_{t \in \mathcal{X}_a(\tau) \wedge t.j=j} t.v$, and $\mathbf{b}[j] = \sum_{t \in \mathcal{X}_b(\tau) \wedge t.j=j} t.v$. Observe that, after executing Update for all $t \in \mathcal{X}_a(\tau)$ and $t \in \mathcal{X}_b(\tau)$ at M_a and M_b, respectively, $\mathcal{S}_a = \sum_{j=1}^n k^j \cdot \mathbf{a}[j]$ and $\mathcal{S}_b = \sum_{j=1}^n k^{n-j+1} \cdot \mathbf{b}[j]$. Moreover, the proof output by Combine is

$$\begin{aligned}\pi_\tau &= g^{(\sum_{j=1}^n k^j \cdot k^{n-j+1} \cdot \mathbf{a}[j] \cdot \mathbf{b}[j]) + r_{a,\tau} \cdot r_{b,\tau} - \rho_{a,\tau} - \rho_{b,\tau}} \\ &= g^{k^{n+1} \cdot (\mathbf{a} \cdot \mathbf{b}) + r_{a,\tau} \cdot r_{b,\tau} - \rho_{a,\tau} - \rho_{b,\tau}}.\end{aligned}$$

If res_τ is equal to $\mathbf{a} \cdot \mathbf{b}$, then the π computed in Verify is equal to π_τ and thus Verify outputs **Yes**. This concludes our proof. □

THEOREM 4.6. *If F is a PRF, then* DDP *is secure under the n-DHE assumption.*

For the formal proof, see Appendix A.3.

Performance. In this scheme, the owner has to invest in some one-time preprocessing effort to create *pub*. This accounts for $O(n)$ exponentiations in $\mathbb{Z}_p$ (for k^i) and another $O(n)$ exponentiations in $\mathbb{G}$ (for g^{k^i}). Nevertheless, this cost is amortized over the entire lifetime of the system. The memory consumption and the computational cost of Sign at each machine are (asymptotically) the same as in DVS, while the cost in Update now involves a modular exponentiation. Note, though, that the latter is performed in the small finite field $\mathbb{Z}_p$ and hence is lightweight.

To analyze the server's computation cost in Combine, first observe that, by setting $i + (n - j + 1) = z$, the server can calculate

$$\prod_{i,j \in [n] \wedge i \neq j} \left(g^{k^{i+(n-j+1)}}\right)^{\mathbf{c}[i][j]} = \prod_{z \in [2n] \setminus \{n+1\}} \left(g^{k^z}\right)^{\sum_{i+(n-j+1)=z} \mathbf{c}[i][j]},$$

assuming it has access to the set of outer product values $\mathbf{c}$. However, the server does not need to explicitly generate $\mathbf{c}$, but rather only needs the vector of n different $\sum_{i+(n-j+1)=z} \mathbf{c}[i][j]$ values, for $2 \leq z \leq 2n$. We can compute these values from the (discrete) convolution of the vectors $\mathbf{a}, \mathbf{b}$, in time $O(n \log n)$ via the fast Fourier transform (FFT). Assuming we have these, then the cost at the server is dominated by $O(n)$ exponentiations in $\mathbb{G}$. For storage, the server has to store the $O(n)$ values in *pub*, which is comparable to the cost of storing $\mathcal{X}_a$ and $\mathcal{X}_b$.

Finally, Verify at the client simply involves a constant number of evaluations of F and a (single) exponentiation in $\mathbb{G}$.

5. NONCE CHAINING TO REDUCE VERIFICATION COST

In this section we present a novel technique, called *nonce chaining*, that can be used to reduce the verification cost at the client in several constructions. Specifically, it is a method employed by the machines upon signing their summaries, which effectively decreases the number of nonces integrated in the final proof to constant when the server combines the signatures together. As a result, the client always needs to reconstruct a constant number of nonces, which minimizes the verification cost. In Section 5.1 we present our nonce chaining technique in detail and rigorously prove its security. In Sections 5.2 and 5.3 we show how this technique is adapted in DVS and DMP, creating optimized schemes DVS* and DMP*, respectively. As a final remark, it is important to stress that nonce chaining is a general technique that does not limit its applicability to the aforesaid schemes. In fact, we materialize this concept with several other specific uses in subsequent sections as well.

5.1. The Nonce Chaining Technique

At a high level, this method views the produced signatures as an ordered list (i.e., a *sequence*) of values. It then modifies the Sign algorithm (performed by the data owner) such that each signature σ in the sequence incorporates: (i) a nonce that does not appear in any signature preceding σ in the sequence, and (ii) the negation of a constant number of terms that depend on nonces that do not appear in σ, or any other signature succeeding σ in the sequence. The effect is that, when all the signatures in the sequence are aggregated together (to produce the final proof), the nonces from the intermediate signatures are eliminated, leaving only those injected by the first and last signatures in the sequence. Since each signature integrates a constant number of nonces, the number of nonces incorporated in the final proof is constant (instead of linear in the number of aggregated signatures), effectively reducing the verification time for the client. Furthermore, we ensure that, if any intermediate signature is removed from the sequence, the previous property ceases to hold. In this sense, the nonces of the intermediate signatures *chain* the first and last signatures.[2]

The security of the nonce chaining technique stems from the fact that each signature incorporates a nonce that does not appear in any signature on its left side in the chain. Informally, this means that, if all the signatures of such a chain are aggregated in any way to produce a proof, then it is guaranteed that at least one nonce from the last signature can never be eliminated, which ensures that the adversary cannot manufacture a proof for a false result. More formally, the security of our schemes modified to use nonce chaining (described next) relies on the following lemma (all operations and elements are in $\mathbb{Z}_p$, where p is a prime whose size is determined by security parameter s).

LEMMA 5.1. *Let $(\sigma_1, \sigma_2, \ldots, \sigma_\ell)$ be a sequence of signatures, where ℓ is a positive integer. Moreover, for every $i \in [\ell]$, let $\sigma_i = \mathbf{k} \cdot \mathbf{v}_i + P(R) + r_i$, $\mathbf{k}$ is a random unknown key vector of size n, $\mathbf{v}_i$ is a known n-element vector, P is some multivariate polynomial on a set R of random unknown nonces that do not appear in $\{\sigma_i, \ldots, \sigma_\ell\}$, and r_i is an unknown random nonce that does not appear in $\{\sigma_1, \ldots, \sigma_{i-1}\}$. Then, an adversary possessing $\{\sigma_1, \sigma_2, \ldots, \sigma_\ell\}$ and $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_\ell\}$ can compute dot product $\mathbf{k} \cdot \mathbf{x}$, for some known n-element vector $\mathbf{x}$, with probability $\mathsf{negl}(s)$.*

PROOF. The adversary can guess $\mathbf{k}$ from any element in $\{\sigma_1, \sigma_2, \ldots, \sigma_\ell\}$ with probability $\mathsf{negl}(s)$ due to Shoup [1997, Lemma 1]. Therefore, the adversary can produce $\mathbf{k} \cdot \mathbf{x}$, with nonnegligible probability only by performing a set of linear operations (addition, subtraction, scalar multiplication) on elements from $\{\sigma_1, \sigma_2, \ldots, \sigma_\ell, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_\ell\}$. We will prove that this cannot happen by contradiction.

Suppose the adversary can find a set $S \subseteq \{\sigma_1, \sigma_2, \ldots, \sigma_\ell\}$ such that it can produce $\mathbf{k} \cdot \mathbf{x}$ by computing $\sum_{\sigma_i \in S} \sigma_i \cdot u_i + \sum_{i \in [\ell]} \mathbf{y}_i \cdot \mathbf{v}_i + d$, where each u_i is a nonzero known scalar, $\mathbf{y}_i$ is a known vector, and d is a known value in $\mathbb{Z}_p$. Let $\sigma_j \in S$ be the signature such that j is the largest index in S, that is, σ_j is the rightmost signature in sequence $(\sigma_1, \sigma_2, \ldots, \sigma_\ell)$ among those in S. By the definition of σ_j, nonce r_j does not appear as a term in any of the signatures in $S \setminus \{\sigma_j\}$ and, consequently, $\sum_{\sigma_i \in S} \sigma_i \cdot u_i + \sum_{i \in [\ell]} \mathbf{y}_i \cdot \mathbf{v}_i + d$ must contain term $u_j \cdot r_j$. Hence, in order for $\sum_{\sigma_i \in S} \sigma_i \cdot u_i + \sum_{i \in [\ell]} \mathbf{y}_i \cdot \mathbf{v}_i + d = \mathbf{k} \cdot \mathbf{x}$ to hold, u_j should be equal to zero. However, we hypothesized that u_j is nonzero, which reaches our contradiction. □

In order to understand the practical importance of this lemma, observe that the signatures in all the schemes we presented so far are of the form $\sigma_i = \mathbf{k} \cdot \mathbf{v}_i + \psi(R) + r_i$. Therefore, if the adversary could compute $\mathbf{k} \cdot \mathbf{x}$ with nonnegligible probability, then he could inject it to any valid signature, inflating or deflating the result value by any factor of $\mathbf{x}$ at will.

In the next two sections, we devise two optimized constructions, called DVS* and DMP*, by augmenting DVS and DMP, respectively, with our nonce chaining technique.

[2]Note that our nonce chaining technique is different from the signature chaining concept proposed in Narasimha and Tsudik [2006]. The latter work binds adjacent record hashes by signing them with an existing public-key digital signature scheme, in order to authenticate a range query result in static outsourced databases under the untrusted client model.

```
// Algorithms KeyGen, Update and Combine are the same as in the DVS construction (Figure 2)

Sign($i, sk, \mathcal{S}_i, \tau$)
1. $r_{i-1,\tau} = F_k(\text{“machine”} \| i-1 \| \text{“epoch”} \| \tau)$
2. $r_{i,\tau} = F_k(\text{“machine”} \| i \| \text{“epoch”} \| \tau)$
3. $\sigma_{i,\tau} = \mathcal{S}_i - r_{i-1,\tau} + r_{i,\tau}$
4. Output $\sigma_{i,\tau}$

Verify($sk, \pi_\tau, res_\tau, \tau$)
1. Parse $sk$ as $k$, and $res_\tau$ as a $n$-element vector
2. $r_{0,\tau} = F_k(\text{“machine”} \| 0 \| \text{“epoch”} \| \tau)$
3. $r_{m,\tau} = F_k(\text{“machine”} \| m \| \text{“epoch”} \| \tau)$
4. Initialize $\pi = -r_{0,\tau} + r_{m,\tau}$
5. For $j = 1$ to $n$
6.     $k_j = F_k(\text{“element”} \| j)$
7.     $\pi = \pi + k_j \cdot res_\tau[j]$
8. If $\pi = \pi_\tau$ output **Yes**, otherwise **No**
```

Fig. 5. The DVS* construction.

$$\begin{aligned}
\pi_\tau &= \sigma_{1,\tau} + \sigma_{2,\tau} + \cdots + \sigma_{m,\tau}\\
\pi_\tau &= (\mathcal{S}_1 - r_{0,\tau} + \cancel{r_{1,\tau}}) + (\mathcal{S}_2 - \cancel{r_{1,\tau}} + \cancel{r_{2,\tau}}) + \cdots + (\mathcal{S}_m - \cancel{r_{m-1,\tau}} + r_{m,\tau})\\
\pi_\tau &= \sum_{i=1}^{m} res_\tau[i] \cdot k_i - r_{0,\tau} + r_{m,\tau}
\end{aligned}$$

Fig. 6. Illustration of the effect of nonce chaining in DVS*: all but a constant number of nonce terms cancel out in the summation.

5.2. The DVS* Construction

This scheme differs from DVS only in the Sign and Verify algorithms, whose pseudocode is presented in Figure 5. At machine M_i, as well as adding nonce $r_{i,\tau}$ as in DVS, procedure Sign now subtracts nonce $r_{i-1,\tau}$ that is correspondingly added to the signature produced by machine M_{i-1} in order to create a nonce chain. Then, the proof produced in Combine (by simply summing up the machines' signatures) now incorporates only two nonces (instead of m), namely $r_{0,\tau}$ and $r_{m,\tau}$, as illustrated in Figure 6. These two nonces are reconstructed by Verify at the client and combined with the result and the element keys in order to match the server's proof.

Correctness and security. We next explain the correctness and security of DVS*.

THEOREM 5.2. DVS* *is correct.*

PROOF. It results from Figure 6 that the way Sign incorporates the nonces in the signatures causes the proof sent by the server to be $\pi_\tau = \sum_{i=1}^{m} k_j \cdot (\sum_{i=1}^{m} \mathbf{a}_i[j]) - r_{0,\tau} + r_{m,\tau}$. Therefore, due to the appropriate modification in Verify to add $(-r_{0,\tau} + r_{m,\tau})$ to π, it is clear that the algorithm outputs **Yes** if res_τ (sent by the server) is equal to $Q(\bigcup_{i=1}^{m} \mathcal{X}_i(\tau))$ (the actual query result). □

THEOREM 5.3. *If F is a PRF, then* DVS* *is secure.*

The proof is provided in Appendix A.4.

Performance. Compared to DVS, the signing cost in DVS* is negligibly affected by an extra hash operation and modular subtraction. On the other hand, nonce chaining eliminates the need for $O(m)$ hashes for computing the nonces in the Verify algorithm of DVS* (now only two are necessary), reducing the verification cost to $O(n)$ hash operations and modular additions/multiplications.

```
// Algorithms KeyGen, Update and Combine are the same as in the DMP construction (Figure 3)

Sign($a, sk, \mathcal{S}_a, \tau$)
1.  For $j = 1$ to $n$
2.      $r_{a,\tau}[j] = F_k(\text{“machine”}||a||\text{“epoch”}||\tau||\text{“}r\text{”}||j)$
3.      $\rho_{a,\tau}[j] = F_k(\text{“machine”}||a||\text{“epoch”}||\tau||\text{“}\rho\text{”}||j)$
4.      $\rho_{a,\tau}[j-1] = F_k(\text{“machine”}||a||\text{“epoch”}||\tau||\text{“}\rho\text{”}||j-1)$
5.      $r_{b,\tau}[j] = F_k(\text{“machine”}||b||\text{“epoch”}||\tau||\text{“}r\text{”}||j)$
6.      $\sigma_{a,\tau}[j] = [(\mathcal{S}_a[j] + r_{a,\tau}[j]), (\mathcal{S}_a[j] \cdot r_{b,\tau}[j] + \rho_{a,\tau}[j] - \rho_{a,\tau}[j-1])]$
7.      $\sigma_{a,\tau}[j][2] = \sigma_{a,\tau}[j][2] + r_{a,\tau}[j] \cdot r_{b,\tau}[j]$ // This line is executed only by machine $M_a$
8.      Output $\sigma_{a,\tau}$

Verify($sk, \pi_\tau, res_\tau, \tau$)
1.  Parse $sk$ as $k$, and $res_\tau$ as a $n_a \times n_b$ matrix
2.  $\rho_{a,\tau}[0] = F_k(\text{“machine”}||a||\text{“epoch”}||\tau||\text{“}\rho\text{”}||0)$
3.  $\rho_{a,\tau}[n] = F_k(\text{“machine”}||a||\text{“epoch”}||\tau||\text{“}\rho\text{”}||n)$
4.  $\rho_{b,\tau}[0] = F_k(\text{“machine”}||b||\text{“epoch”}||\tau||\text{“}\rho\text{”}||0)$
5.  $\rho_{b,\tau}[n] = F_k(\text{“machine”}||b||\text{“epoch”}||\tau||\text{“}\rho\text{”}||n)$
6.  Initialize $\pi = \rho_{a,\tau}[0] + \rho_{b,\tau}[0] - \rho_{a,\tau}[n] - \rho_{b,\tau}[n]$
7.  For $i = 1$ to $n_a$, $k_{a,i} = F_k(\text{“machine”}||a||\text{“element”}||i)$
8.  For $j = 1$ to $n_b$, $k_{b,j} = F_k(\text{“machine”}||b||\text{“element”}||j)$
9.  $\pi = \pi + \sum_{i\in[n_a],j\in[n_b]} k_{a,i} \cdot k_{b,j} \cdot res_\tau[i][j]$
10. If $\pi = \pi_\tau$ output Yes, otherwise No
```

Fig. 7. The DMP* construction.

5.3. The DMP* Construction

Similar to the case of DVS*, DMP* differs from its counterpart DMP in the Sign and Verify operations, which are presented in Figure 7. Note that the routines in pseudocode correspond only to machine M_a. The case for M_b is symmetric, with the only difference being in line 7 of Sign, which is executed only by M_a. Whereas DVS* applied the nonce chaining technique on the nonces created by different machines, DMP* uses it on those generated for different signature components. Recall that, in DMP, every machine signature is an n-element vector, where each element is comprised of two components. The nonce negations are injected to the second component of every one of the n elements of each signature. The effect of nonce chaining in DMP* is demonstrated in Figure 8. After applying the cancellations, there are only a constant number of nonces in the proof sent by the server to the client. The latter reconstructs these nonces in Verify and combines them with the result vector and the element keys in order to match the server's proof.

Correctness and security. The following two theorems state the correctness and security of DMP*, respectively.

THEOREM 5.4. DMP* *is correct.*

PROOF. From Figure 8, the way Sign incorporates the nonces in the signatures causes the proof sent by the server to be $\pi_\tau = \sum_{i\in[n_a],j\in[n_b]} k_{a,i} \cdot k_{b,j} \cdot res_\tau[i][j] + \rho_{a,\tau}[0] + \rho_{b,\tau}[0] - \rho_{a,\tau}[n] - \rho_{b,\tau}[n]$. Therefore, due to the appropriate modification in Verify to add $\rho_{a,\tau}[0] + \rho_{b,\tau}[0] - \rho_{a,\tau}[n] - \rho_{b,\tau}[n]$ to π, it is clear that the algorithm outputs **Yes** if res_τ (sent by the server) is equal to $Q(\mathcal{X}_a(\tau) \cup \mathcal{X}_b(\tau)) = \mathbf{AB}$ (the actual query result). □

THEOREM 5.5. *If F is a PRF, then* DMP* *is secure.*

The proof is similar to previous security proofs and is given in Appendix A.5.

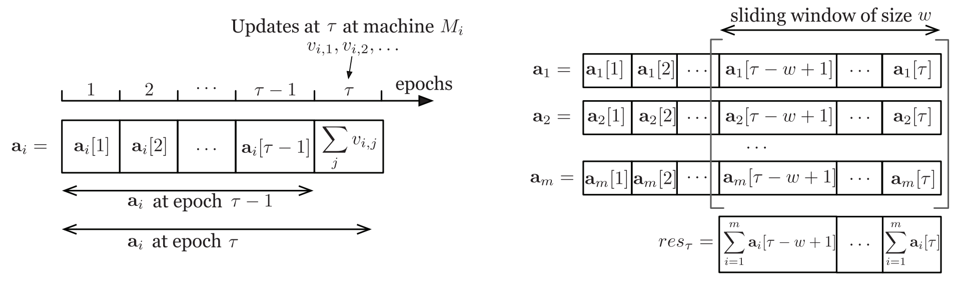
Performance. Compared to DMP, the signing cost in DMP* is negligibly affected by an extra hash operation and modular subtraction (and, in the case of M_a, an additional

$$
\begin{aligned}
\pi_\tau = \;& \sigma_{a,\tau}[1][1]\cdot\sigma_{b,\tau}[1][1] - \sigma_{a,\tau}[1][2] - \sigma_{b,\tau}[1][2] + \\
& \sigma_{a,\tau}[2][1]\cdot\sigma_{b,\tau}[2][1] - \sigma_{a,\tau}[2][2] - \sigma_{b,\tau}[2][2] + \\
& \cdots \\
& \sigma_{a,\tau}[n][1]\cdot\sigma_{b,\tau}[n][1] - \sigma_{a,\tau}[n][2] - \sigma_{b,\tau}[n][2]
\end{aligned}
$$

$$
\begin{aligned}
\pi_\tau = \;& \sum_{i\in[n_a],j\in[n_b]} k_{a,i}\cdot k_{b,j}\cdot res_\tau[i][j] + \\
& \cancel{r_{a,\tau}[1]\cdot r_{b,\tau}[1]} - \cancel{\rho_{a,\tau}[1]} + \rho_{a,\tau}[0] - \cancel{\rho_{b,\tau}[1]} + \rho_{b,\tau}[0] - \cancel{r_{a,\tau}[1]\cdot r_{b,\tau}[1]} + \\
& \cancel{r_{a,\tau}[2]\cdot r_{b,\tau}[2]} - \cancel{\rho_{a,\tau}[2]} + \cancel{\rho_{a,\tau}[1]} - \cancel{\rho_{b,\tau}[2]} + \cancel{\rho_{b,\tau}[1]} - \cancel{r_{a,\tau}[2]\cdot r_{b,\tau}[2]} + \\
& \cdots \\
& \cancel{r_{a,\tau}[n]\cdot r_{b,\tau}[n]} - \rho_{a,\tau}[n] + \cancel{\rho_{a,\tau}[n-1]} - \rho_{b,\tau}[n] + \cancel{\rho_{b,\tau}[n-1]} - \cancel{r_{a,\tau}[n]\cdot r_{b,\tau}[n]}
\end{aligned}
$$

$$
\pi_\tau = \sum_{i\in[n_a],j\in[n_b]} k_{a,i}\cdot k_{b,j}\cdot res_\tau[i][j] + \rho_{a,\tau}[0] + \rho_{b,\tau}[0] - \rho_{a,\tau}[n] - \rho_{b,\tau}[n]
$$

Fig. 8. Illustration of the effect of nonce chaining in DMP*.



(a) updates (b) result

Fig. 9. The model for dynamic vector sum over a sliding window.

modular multiplication and subtraction). On the other hand, nonce chaining eliminates the need for $O(n)$ hashes for computing the nonces in the Verify algorithm of DMP* (now only four are necessary), reducing the verification cost to $O(n_a + n_b)$ hash operations and $O(n_a n_b)$ modular multiplications and additions. Note that this can have significant impact on the costs when n is large compared to n_a and n_b, as may commonly be the case in practice (when n_a and n_b correspond to a small number of entities that are each associated with high-dimensional rows/columns of feature values).

6. SLIDING WINDOW QUERIES

Sections 6.1–6.3 extend our DVS*, DMP*, and DDP constructions, respectively, to the sliding window setting. Section 6.4 explains how to handle multiple concurrent sliding window queries with variable parameters.

6.1. Dynamic Vector Sum Authentication over a Sliding Window

We consider m machines M_i. At every epoch τ, each machine generates updates of the form (τ, v) that are sent to the server. Conceptually, the stream $\mathcal{X}_i$ of M_i produces a dynamic vector $\mathbf{a}_i$ that is expanded at every epoch τ by one element as shown in Figure 9(a). Let $\mathbf{a}_i = [\mathbf{a}_i[1], \mathbf{a}_i[2], \ldots, \mathbf{a}_i[\tau-1]]$ be the (conceptual) vector of M_i at the end of epoch $\tau-1$. Then, this vector becomes $\mathbf{a}_i = [\mathbf{a}_i[1], \mathbf{a}_i[2], \ldots, \mathbf{a}_i[\tau-1], \sum_j v_{i,j}]$ at the end of epoch τ, where $v_{i,j}$ corresponds to the updates generated at τ by M_i.

Given a long-running sliding window query Q with window size w from the client, the server computes and sends to the client at *every* epoch τ the value

$$Q\left(\bigcup_{i=1}^{m} \mathcal{X}_i(\tau)\right) = \sum_{i=1}^{m} \mathbf{a}_i[\tau - w + 1 : \tau] = \left[\sum_{i=1}^{m} \mathbf{a}_i[\tau - w + 1], \ldots, \sum_{i=1}^{m} \mathbf{a}_i[\tau]\right],$$

that is, the sum of the vectors comprised of the last w elements of the machines' (conceptual) vectors. This is visualized in Figure 9(b).

The first step towards reaching our main result begins by adopting a general methodology briefly mentioned in Nath and Venkatesan [2013] for the untrusted client model, which, however is applicable to our setting as well. This method simply applies an existing authentication scheme independently at every epoch as a black box, but aggregates the w signatures that correspond to the current sliding window, allowing for a single proof verification. For efficiency of verification, we also adopt nonce chaining: in our setting, the machines run DVS* independently at every epoch, that is, they collectively authenticate $[\sum_{i=1}^{m} \mathbf{a}_i[\tau]]$ at every τ with DVS*, as if it is the vector sum of singleton vectors $[\mathbf{a}_1[\tau]], \ldots, [\mathbf{a}_m[\tau]]$. To do so, every machine M_i sends to the server signature $\sigma_{i,\tau} = k_\tau \cdot \sum_j v_{i,j} - r_{i-1,\tau} + r_{i,\tau}$, where $v_{i,j}$ refers to the updates occurring at τ at M_i, k_τ is an epoch-dependent key, and $r_{i,\tau}, r_{i-1,\tau}$ are epoch- and machine-dependent random nonces. Values $k_\tau, r_{i,\tau}$, and $r_{i-1,\tau}$ are all produced with the master secret key sk. At every epoch τ, the server sends to the client a proof that is the summation of the signatures $\sigma_{i,j}$, where $i \in [m]$ and $j \in \{\tau - w + 1, \ldots, \tau\}$, that is, of the signatures of all the machines corresponding to the elements lying within the current sliding window of size w. Observe this proof is equal to $\pi_\tau = k_{\tau-w+1} \cdot res_\tau[1] + \cdots + k_\tau \cdot res_\tau[w] + \sum_{j=\tau-w+1}^{\tau}(r_{m,j} - r_{0,j})$, where $res_\tau = Q(\bigcup_{i=1}^{m} \mathcal{X}_i(\tau))$. Since the client possesses sk, it can produce all the keys and nonces and eventually verify the result.

We further optimize the verification process of the preceding scheme by applying nonce chaining across subsequent epochs as well as across machines. Specifically, we eliminate the summation $\sum_{j=\tau-w+1}^{\tau}(r_{m,j} - r_{0,j})$, which involves $O(w)$ nonces, resulting in a proof that contains only a constant number of nonces. This is achieved by forcing the m^{th} machine at τ to add $(r_{0,\tau-1} - r_{m,\tau-1})$ to its signature (shown in line 7 of Sign in Figure 10). Although, asymptotically, the total verification overhead remains $O(w)$, this optimization saves $O(w)$ PRF evaluations and modular additions. We provide the details of our complete scheme, called W-DVS (after sliding Window DVS), in Figure 10.

Correctness and security. In the following two theorems, we prove the correctness and security of W-DVS, respectively.

THEOREM 6.1. W-DVS *is correct.*

PROOF. Let the actual result of Q at τ be

$$Q\left(\bigcup_{i=1}^{m} \mathcal{X}_i(\tau)\right) = \left[\sum_{i=1}^{m} \mathbf{a}_i[\tau - w + 1], \ldots, \sum_{i=1}^{m} \mathbf{a}_i[\tau]\right],$$

where $\mathbf{a}_i[j] = \sum_{t \in \mathcal{X}_i(\tau) \wedge t.\tau = j} t.v$. After executing Update for all $t \in \mathcal{X}_i(\tau)$ generated at epoch τ at any M_i, the summary becomes $\mathcal{S}_i = k_\tau \cdot \mathbf{a}_i[\tau]$. Therefore, running Sign at τ, M_i produces signature $\sigma_{i,\tau} = k_\tau \cdot \mathbf{a}_i[\tau] - r_{i-1,\tau} + r_{i,\tau}$ for $i \in [m-1]$, whereas M_m generates $\sigma_{m,\tau} = k_\tau \cdot \mathbf{a}_m[\tau] - r_{m-1,\tau} + r_{m,\tau} + r_{0,\tau-1} - r_{m,\tau-1}$. Moreover, observe that

```
KeyGen$(1^s)$
1.  $k \xleftarrow{\$} \mathbb{Z}_p^*$
2.  Output $sk = k$ and $pub = \perp$

Update$(i, sk, \mathcal{S}_i, t)$
1.  If $t$ is the first update in this epoch, $\mathcal{S}_i = 0$
2.  Parse $t$ as $(\tau, v)$, and $sk$ as $k$
3.  $k_\tau = F_k(\text{“element”} \| \tau)$
4.  $\mathcal{S}_i = \mathcal{S}_i + k_\tau \cdot v$
5.  Output $\mathcal{S}_i$

Sign$(i, sk, \mathcal{S}_i, \tau)$
1.  $r_{i-1,\tau} = F_k(\text{“machine”} \| i-1 \| \text{“epoch”} \| \tau)$
2.  $r_{i,\tau} = F_k(\text{“machine”} \| i \| \text{“epoch”} \| \tau)$
3.  $\sigma_{i,\tau} = \mathcal{S}_i - r_{i-1,\tau} + r_{i,\tau}$
4.  If $i = m$
5.      $r_{0,\tau-1} = F_k(\text{“machine”} \| 0 \| \text{“epoch”} \| \tau - 1)$
6.      $r_{m,\tau-1} = F_k(\text{“machine”} \| m \| \text{“epoch”} \| \tau - 1)$
7.      $\sigma_{i,\tau} = \sigma_{i,\tau} + r_{0,\tau-1} - r_{m,\tau-1}$
8.  Output $\sigma_{i,\tau}$

Combine$(\bigcup_{i \in [m], j \in \{\tau-w+1,\dots,\tau\}} \{\sigma_{i,j}\}, \emptyset, \perp)$
1.  Output $\pi_\tau = \sum_{i=1}^{m} \sum_{j=\tau-w+1}^{\tau} \sigma_{i,j}$

Verify$(sk, \pi_\tau, res_\tau, \tau)$
1.  Parse $sk$ as $k$, and $res_\tau$ as a $w$-element vector
2.  $r_{0,\tau-w} = F_k(\text{“machine”} \| 0 \| \text{“epoch”} \| \tau - w)$
3.  $r_{m,\tau-w} = F_k(\text{“machine”} \| m \| \text{“epoch”} \| \tau - w)$
4.  $r_{0,\tau} = F_k(\text{“machine”} \| 0 \| \text{“epoch”} \| \tau)$
5.  $r_{m,\tau} = F_k(\text{“machine”} \| m \| \text{“epoch”} \| \tau)$
6.  Initialize $\pi = r_{0,\tau-w} - r_{m,\tau-w} + r_{m,\tau} - r_{0,\tau})$
7.  For $j = (\tau - w + 1)$ to $\tau$
8.      $k_j = F_k(\text{“element”} \| j)$
10.     $\pi = \pi + k_j \cdot res_\tau[j]$
11. If $\pi = \pi_\tau$ output **Yes**, otherwise **No**
```

Fig. 10. The W-DVS construction.

$\sum_{i=1}^{m} \sigma_{i,\tau} = (k_\tau \cdot \sum_{i=1}^{m} \mathbf{a}_i[\tau]) - r_{0,\tau} + r_{m,\tau} + r_{0,\tau-1} - r_{m,\tau-1}$. Hence, Combine outputs proof

$$\pi_\tau = \sum_{i=1}^{m} \sum_{j=\tau-w+1}^{\tau} \sigma_{i,j} = \left(\sum_{j=\tau-w+1}^{\tau} k_j \cdot \sum_{i=1}^{m} \mathbf{a}_i[j] \right) + r_{0,\tau-w} - r_{m,\tau-w} + r_{m,\tau} - r_{0,\tau}.$$

Now notice that, if res_τ passed in Verify is equal to $Q(\bigcup_{i=1}^{m} \mathcal{X}_i(\tau))$, then the algorithm computes a π that is equal to π_τ and thus the output is **Yes**. □

THEOREM 6.2. *If F is a PRF, then* W-DVS *is secure.*

Proof of security is shown in Appendix A.6.

Performance. Similar to DVS, the storage, computational time of Update and Sign, and communication cost at each machine are all constant. At the client, the storage cost is constant, the verification cost entails $O(w)$ PRF evaluations and modular additions/multiplications, whereas the communication cost is constant due to the constant-sized proof generated by the server. Regarding the server's computational cost, note that in Figure 10 we present a simplified version of Combine for the sake of clarity. This naive algorithm runs in $O(m \cdot w)$ time. However, we can improve upon this cost by

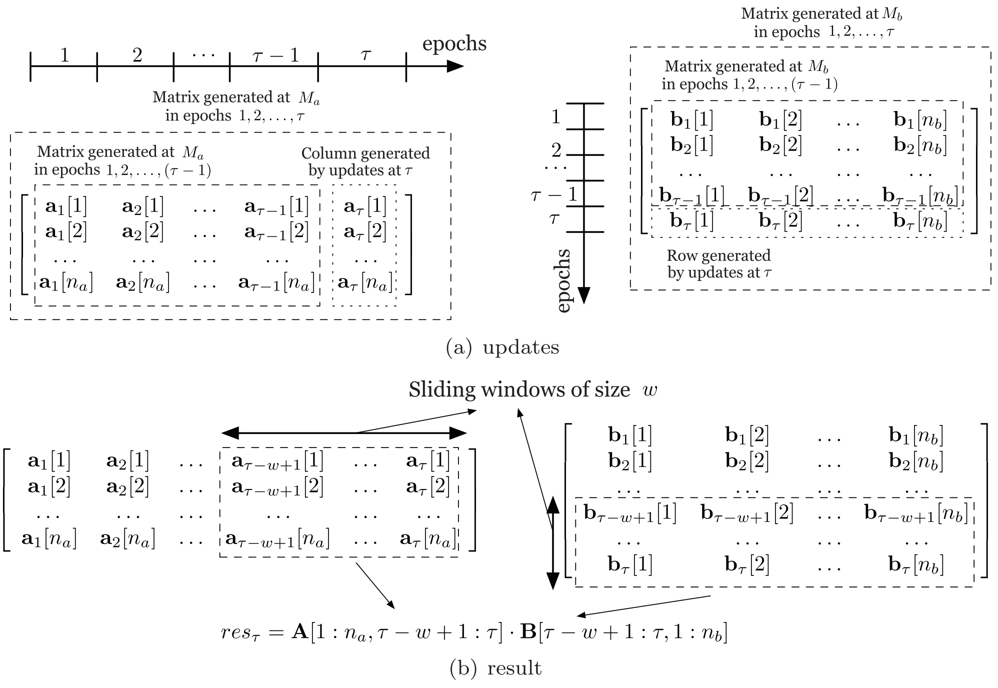

epochs
Matrix generated at M_a in epochs $1, 2, \ldots, \tau$
Matrix generated at M_a in epochs $1, 2, \ldots, (\tau-1)$
Column generated by updates at τ
Matrix generated at M_b in epochs $1, 2, \ldots, \tau$
Matrix generated at M_b in epochs $1, 2, \ldots, (\tau-1)$
Row generated by updates at τ
(a) updates
Sliding windows of size w
$res_\tau = \mathbf{A}[1 : n_a, \tau - w + 1 : \tau] \cdot \mathbf{B}[\tau - w + 1 : \tau, 1 : n_b]$
(b) result

Fig. 11. The model for dynamic matrix product over a sliding window.

having the server maintain a simple structure that requires $O(w)$ space. Specifically, at the end of every epoch $\tau - 1$, the server maintains an ordered list of the sequence of values $(\sum_{i=1}^{m} \sigma_{i,\tau-w}, \ldots, \sum_{i=1}^{m} \sigma_{i,\tau-1})$, where $\sum_{i=1}^{m} \sigma_{i,\tau-w}$ is at the tail and $\sum_{i=1}^{m} \sigma_{i,\tau-1}$ at the head of the list. Moreover, it stores proof $\pi_{\tau-1} = \sum_{i=1}^{m} \sum_{j=\tau-w}^{\tau-1} \sigma_{i,j}$. At the next epoch τ, the server: (i) receives $\sigma_{i,\tau}$ for all $i \in [m]$ and computes $\sum_{i=1}^{m} \sigma_{i,\tau}$, (ii) removes $\sum_{i=1}^{m} \sigma_{i,\tau-w}$ from the tail of the list, (iii) adds $\sum_{i=1}^{m} \sigma_{i,\tau}$ at the head of the list, and finally (iv) computes the proof for epoch τ as $\pi_\tau = \pi_{\tau-1} - \sum_{i=1}^{m} \sigma_{i,\tau-w} + \sum_{i=1}^{m} \sigma_{i,\tau}$. With the previous modification the total computational cost of Combine decreases to $O(m)$ modular additions.

6.2. Dynamic Matrix Product Authentication over a Sliding Window

We assume two machines M_a and M_b. At epoch τ, M_a generates a new n_a-element column vector $\mathbf{a}_\tau$. Conceptually, the stream $\mathcal{X}_a$ of M_a generates a dynamic matrix $\mathbf{A}$ that expands its number of columns by one at each new epoch as follows: let $\mathbf{a}_i$ be an n_a-element vector, and $\mathbf{A} = [\mathbf{a}_1\mathbf{a}_2 \ldots \mathbf{a}_{\tau-1}]$ the $n_a \times (\tau - 1)$ (conceptual) matrix of M_a at epoch $\tau - 1$. At epoch τ, M_a creates a new $n_a \times \tau$ matrix $\mathbf{A} = [\mathbf{a}_1\mathbf{a}_2 \ldots \mathbf{a}_{\tau-1}\mathbf{a}_\tau]$ by appending an n_a-element vector $\mathbf{a}_\tau$ as the rightmost column of the matrix, where $\mathbf{a}_\tau$ is generated by the updates occurring at τ. Similarly, at epoch τ, M_b produces a new n_b-element row vector $\mathbf{b}_\tau$. Conceptually, the stream $\mathcal{X}_b$ of M_b generates a dynamic matrix $\mathbf{B}$ that expands its number of rows by one at each new epoch as follows: let $\mathbf{b}_i$ be an n_b-element vector, and $\mathbf{B} = [\mathbf{b}_1\mathbf{b}_2 \ldots \mathbf{b}_{\tau-1}]^{\mathrm{T}}$ be the $(\tau - 1) \times n_b$ (conceptual) matrix of M_b at epoch $\tau - 1$. At epoch τ, M_b creates a new $\tau \times n_b$ matrix $\mathbf{B} = [\mathbf{b}_1\mathbf{b}_2 \ldots \mathbf{b}_{\tau-1}\mathbf{b}_\tau]^{\mathrm{T}}$ by appending the newly arrived $\mathbf{b}_\tau$ at the bottom of the matrix. This window model matches many natural motivating scenarios that demand matrix product; see Section 8.5 for examples. The update process is illustrated in Figure 11(a).

```
KeyGen($1^s$)
1. $k \xleftarrow{\$} \mathbb{Z}_p^*$
2. Output $sk = k$ and $pub = \perp$

Update($a, sk, \mathcal{S}_a, t$)
1. Parse $t$ as $(i, \tau, v)$, and $sk$ as $k$
2. If $t$ is the first update in this epoch, $\mathcal{S}_a = 0$
3. $k_{a,i} = F_k(\text{“machine”}\|a\|\text{“element”}\|i)$
4. $\mathcal{S}_a = \mathcal{S}_a + k_{a,i} \cdot v$
5. Output $\mathcal{S}_a$

Sign($a, sk, \mathcal{S}_a, \tau$)
1. $r_{a,\tau} = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau\|\text{“}r\text{”})$
2. $r_{b,\tau} = F_k(\text{“machine”}\|b\|\text{“epoch”}\|\tau\|\text{“}r\text{”})$
3. $\rho_{a,\tau} = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau\|\text{“}\rho\text{”})$
4. $\rho_{a,\tau-1} = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau-1\|\text{“}\rho\text{”})$
5. $\sigma_{a,\tau} = [(\mathcal{S}_a + r_{a,\tau}), (\mathcal{S}_a \cdot r_{b,\tau} + \rho_{a,\tau} - \rho_{a,\tau-1})]$
6. $\sigma_{a,\tau}[2] = \sigma_{a,\tau}[2] + r_{a,\tau} \cdot r_{b,\tau}$ // This line is executed only by machine $M_a$
7. Output $\sigma_{a,\tau}$

Combine($\bigcup_{j\in\{\tau-w+1,\ldots,\tau\}}\{\sigma_{a,j}, \sigma_{b,j}\}, \emptyset, \perp$)
1. $\pi_\tau = \sum_{j=\tau-w+1}^{\tau}(\sigma_{a,j}[1] \cdot \sigma_{b,j}[1] - \sigma_{a,j}[2] - \sigma_{b,j}[2])$
2. Output $\pi_\tau$

Verify($sk, \pi_\tau, res_\tau, \tau$)
1. Parse $sk$ as $k$, and $res_\tau$ as a $n_a \times n_b$ matrix
2. $\rho_{a,\tau-w} = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau-w\|\text{“}\rho\text{”})$
3. $\rho_{a,\tau} = F_k(\text{“machine”}\|a\|\text{“epoch”}\|\tau\|\text{“}\rho\text{”})$
4. $\rho_{b,\tau-w} = F_k(\text{“machine”}\|b\|\text{“epoch”}\|\tau-w\|\text{“}\rho\text{”})$
5. $\rho_{b,\tau} = F_k(\text{“machine”}\|b\|\text{“epoch”}\|\tau\|\text{“}\rho\text{”})$
6. Initialize $\pi = \rho_{a,\tau-w} + \rho_{b,\tau-w} - \rho_{a,\tau} - \rho_{b,\tau}$
7. For $i = 1$ to $n_a$, $k_{a,i} = F_k(\text{“machine”}\|a\|\text{“element”}\|i)$
8. For $j = 1$ to $n_b$, $k_{b,j} = F_k(\text{“machine”}\|b\|\text{“element”}\|j)$
9. $\pi = \pi + \sum_{i\in[n_a],j\in[n_b]} k_{a,i} \cdot k_{b,j} \cdot res_\tau[i][j]$
10. If $\pi = \pi_\tau$ output **Yes**, otherwise **No**
```

Fig. 12. The W-DMP construction.

Given a long-running sliding window query Q with window size w from the client, the server computes and sends to the client at each epoch τ the $n_a \times n_b$ matrix

$$\begin{aligned} res_\tau &= \mathbf{A}[1:n_a, \tau-w+1:\tau] \cdot \mathbf{B}[\tau-w+1:\tau, 1:n_b] \\ &= [\mathbf{a}_{\tau-w+1}, \ldots, \mathbf{a}_\tau] \cdot [\mathbf{b}_{\tau-w+1}, \ldots, \mathbf{b}_\tau]^{\mathrm{T}}, \end{aligned}$$

that is, the product between the $n_a \times w$ matrix produced by the last w columns of $\mathbf{A}$, and the $w \times n_b$ matrix derived from the last w rows of $\mathbf{B}$. Figure 11(b) demonstrates this result.

We next present a scheme, called W-DMP, that enables the authentication of the sliding window matrix products defined before. Figure 12 provides the pseudocode of the construction. The main idea is similar to DMP*, namely we apply nonce chaining on the signature components. The difference here is that M_a (M_b) need not maintain $O(w)$ summaries, that is, one for every column (respectively, row) lying in the sliding window. In contrast, they need to maintain only a summary for the current epoch, which is then signed and forwarded to the server at the end of the epoch. The nonce chaining technique helps eliminate the $O(w)$ factor from the verification cost that stems from the computation of the nonces (similar to DMP*, here the server's proof incorporates exactly four nonces).

Correctness and security. Observe that, at any epoch τ, the server in W-DMP possesses $O(w)$ signatures, one for each column (row) of the submatrix of M_a (respectively, M_b) lying in the current window. The keys and nonces for these signatures are constructed in a similar manner to that in DMP*. Therefore, both correctness and security of W-DMP are proven in an identical manner to that in DMP*. We omit the detailed proofs to avoid repetition.

Performance. All costs at the machines are constant. At the server, we employ a similar optimization as in W-DVS to reduce the straightforward $O(w)$ cost to constant, sacrificing $O(w)$ memory space. In particular, to ensure that the cost of Combine is constant, at $\tau - 1$, the server maintains signatures $\sigma_{a,\tau-w}, \sigma_{b,\tau-w}, \ldots, \sigma_{a,\tau-1}, \sigma_{b,\tau-1}$ in a linked list having $\sigma_{a,\tau-w}$ at the tail and $\sigma_{b,\tau-1}$ at the head. Moreover, it stores $\pi_{\tau-1}$. At epoch τ, the server receives $\sigma_{a,\tau}, \sigma_{b,\tau}$, removes $\sigma_{a,\tau-w}, \sigma_{b,\tau-w}$ from the tail of the list, adds $\sigma_{a,\tau}, \sigma_{b,\tau}$ to the head of the list, and computes $\pi_\tau = \pi_{\tau-1} - (\sigma_{a,\tau-w}[1]\cdot\sigma_{b,\tau-w}[1] - \sigma_{a,\tau-w}[2] - \sigma_{b,\tau-w}[2]) + (\sigma_{a,\tau}[1]\cdot\sigma_{b,\tau}[1] - \sigma_{a,\tau}[2] - \sigma_{b,\tau}[2])$. Finally, the verification cost at the client entails $O(n_a + n_b)$ PRF evaluations and $O(n_a n_b)$ modular multiplications/additions.

6.3. Dynamic Dot Product Authentication over a Sliding Window

Suppose there are two machines M_a and M_b. At every epoch τ, each machine generates tuples of the form (τ, v) that are sent to the server. Conceptually, the stream $\mathcal{X}_a$ (respectively, $\mathcal{X}_b$) of M_a (respectively, M_b) produces a dynamic vector $\mathbf{a}$ (respectively, $\mathbf{b}$) which is expanded at every epoch τ by one element exactly as described in W-DVS (see Figure 9(a) in Section 6.1).

Given a long-running sliding window query Q with window size w from the client, the server computes and sends to the client at each epoch τ the value

$$res_\tau = \mathbf{a}[\tau - w + 1 : \tau] \cdot \mathbf{b}[\tau - w + 1 : \tau] = \sum_{i=\tau-w+1}^{\tau} \mathbf{a}[i] \cdot \mathbf{b}[i],$$

that is, the dot product of the vectors comprised of the last w elements of the two machine vectors. We next describe W-DDP that authenticates such results.

Before attempting to devise a new scheme, we make two important observations. First, we notice that we can perceive $\mathbf{a}[\tau - w + 1 : \tau]$ as a $1 \times w$ matrix, where each column is comprised of a single element. Moreover, we can regard $\mathbf{b}[\tau - w + 1 : \tau]^{\mathrm{T}}$ as a $w \times 1$ matrix, where every row is comprised of a single element. Then, the matrix product $\mathbf{a}[\tau - w + 1 : \tau] \cdot \mathbf{b}[\tau - w + 1 : \tau]^{\mathrm{T}}$ is a 1×1 matrix consisting of a single element that is the result res_τ of our dot product query. This is illustrated in Figure 13. Second, we observe that, similar to W-DVS, each machine can produce a signature on the summation of all its update values at every epoch τ. More specifically, at every τ, machine M_a (M_b) can produce a signature on the single-element column (respectively, row) corresponding to τ. The prior two observations suggest that we can authenticate dot products over sliding windows by simply invoking W-DMP on $1 \times w$ matrix $\mathbf{a}[\tau - w + 1 : \tau]$ and $w \times 1$ matrix $\mathbf{b}[\tau - w + 1]^{\mathrm{T}}$.

Observe that the sliding window setting obviates the need of our basic DDP construction to provide the server with public information to remove "unnecessary" components from the final proof. Specifically, contrary to DDP where the machines produce each signature on all the elements of their vectors, here they sign every vector element individually. This is because every vector element corresponds to a unique epoch, which in turn is associated with a unique signature. Therefore, when the server multiplies pairs of signatures corresponding to the same epoch following W-DMP, no "unnecessary" elements are introduced in the proof, that is, the proof incorporates only products of the form $\mathbf{a}[i] \cdot \mathbf{b}[j]$, where it always holds that $i = j$. In this sense, dot product

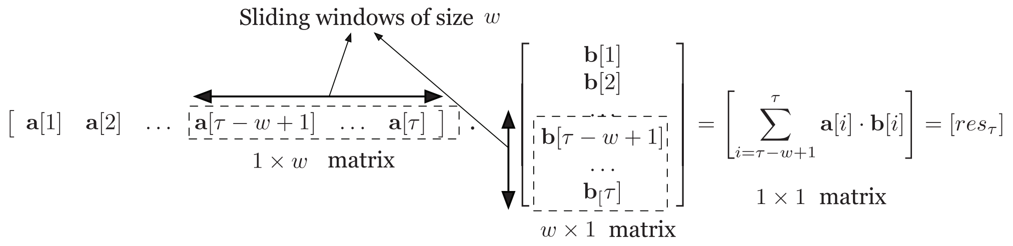

Sliding windows of size w
$[\ \mathbf{a}[1]\ \ \mathbf{a}[2]\ \ \dots\ \mathbf{a}[\tau-w+1]\ \dots\ \mathbf{a}[\tau]\ ]$
$1 \times w$ matrix
$\mathbf{b}[1]$
$\mathbf{b}[2]$
$\mathbf{b}[\tau-w+1]$
$\mathbf{b}[\tau]$
$w \times 1$ matrix
$= \left[\sum_{i=\tau-w+1}^{\tau} \mathbf{a}[i]\cdot\mathbf{b}[i]\right] = [res_\tau]$
1×1 matrix

Fig. 13. Reducing dynamic dot product over a sliding window to W-DMP.

authentication over sliding windows can be performed considerably faster than in the case of DDP.

Correctness and security. The correctness and security of W-DDP follow directly from the correctness and security of W-DMP.

Performance. Similar to W-DMP, all the costs at the machines in W-DDP are constant. Moreover, applying the optimizations discussed for W-DMP, the server requires $O(w)$ space and constant processing time to produce the proof. Finally, the verification cost at the client becomes $O(1)$, since $n_a = n_b = 1$ in the case of dot product.

6.4. Concurrent Queries with Variable Parameters

So far, we have assumed a single query with window size w. Moreover, we have focused on the case where the server must send a result to the client at every epoch. In other words, we fixed the *tumbling factor*, which determines how many epochs the window should slide over the time domain before the server reports a new result, to unity. In this section, we explain how our schemes can be modified to handle multiple concurrent queries with variable window sizes and tumbling factors.

Variable window sizes. Observe that we use the window size w in the Combine and Verify algorithms of all our constructions as a fixed global variable. However, the correctness and security of the schemes hold for an arbitrary w. Consequently, in order to support multiple queries with variable w, we can simply modify Combine and Verify to take w as a user-defined input parameter. Interestingly, it is not necessary to modify the machine's algorithms Update and Sign, as these are independent of w. As such, no extra overhead is imposed at the machines for handling multiple queries in the sliding window case.

Recall that the server can construct the proof of a single sliding window query in $O(1)$ time requiring $O(w)$ space, after having computed in $O(m)$ time the new head of the list from the newly arrived signatures. In the case of N queries with sizes $w_1, w_2, \dots, w_N$, where w_N is the largest size, the server can produce all proofs independently in $O(N)$ time and $O(w_1 + w_2 + \cdots + w_N)$ size. We can improve this space cost to $O(w_N + N)$ with the structure shown in Figure 14. The server maintains a doubly linked list over the aggregated signatures corresponding to the last w_N epochs, as we described earlier for W-DVS and W-DMP. Moreover, it maintains pointers to indicate the end of the windows with sizes $w_1, \dots, w_{N-1}$. It also stores proofs $\pi_{\tau-1,1}, \pi_{\tau-1,2}, \dots, \pi_{\tau-1,N}$, where $\pi_{\tau-1,i}$ corresponds to the proof constructed in epoch $\tau - 1$ for query with size w_i. In the next epoch τ, the server removes the element at the tail (corresponding to epoch $\tau - w_N$) and appends the new signature(s) at the head (corresponding to epoch τ). Subsequently, it updates every $\pi_{\tau-1,i}$ to yield $\pi_{\tau,i}$, making use of the window end pointers. Finally, it shifts all pointers by one place to the right in order to indicate the new end of the query

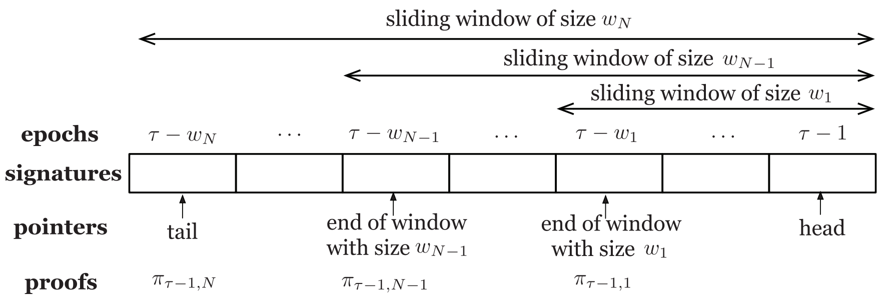

sliding window of size w_N
sliding window of size w_{N-1}
sliding window of size w_1
epochs
$\tau - w_N$
$\tau - w_{N-1}$
$\tau - w_1$
$\tau - 1$
signatures
pointers
tail
end of window with size w_{N-1}
end of window with size w_1
head
proofs
$\pi_{\tau-1,N}$
$\pi_{\tau-1,N-1}$
$\pi_{\tau-1,1}$

Fig. 14. Structure maintained at the server at epoch $\tau - 1$.

windows. Evidently, the total space consumption is $O(w_N)$ for storing the signatures for w_N epochs, plus $O(N)$ for maintaining the proofs and pointers. The total processing cost for the construction of the N proofs is $O(N)$, excluding the $O(m)$ cost for calculating the new head of the list.

Variable tumbling factors. Consider the case where a client sets an arbitrary tumbling factor *tf*, which can be any positive integer. The client then expects to receive a new result and proof from the server every *tf* epochs. Since the client is loosely synchronized with the machines and the server within every epoch, it can easily detect whether the server maliciously omits sending a new result. Let τ be an epoch when the server sends a new result and proof to the client respecting *tf*, and let w be the window size. Observe that both τ and w are parameters that do not affect the correctness and security of our schemes. Therefore, our constructions enable the client to verify the result corresponding to the window of size w at τ, without requiring any modifications.

For the case where there are multiple concurrent queries with variable window sizes and tumbling factors, we can adopt the structure described in Figure 14, slightly modifying it to accommodate information about the tumbling factors. Specifically, for each query, in addition to its end pointer and proof, the server also sets a counter equal to *tf* before the system commences. Then, at every epoch, it operates as we discussed previously for the variable window sizes (i.e., it properly updates the signature list, pointers, and proofs), but also reduces all the counters by 1. The server sends a new result and proof for a query whenever the respective counter becomes 0. The server reinitializes the counters that become 0 to their original *tf* value, and the system continues in the aforesaid fashion.

7. SUBSET QUERIES

So far, in our DVS/DVS*, DMP/DMP*, and DDP schemes, the query involved a linear algebraic computation over the entire vectors and matrices dynamically generated by the machines. In this section we target at an arbitrary number of queries, capturing the case where there are multiple clients (all trusted with the secret key by the data owner). Each client can pose a different query that may require an algebraic computation over any subvectors (in the case of DVS/DVS* and DDP) or submatrices (in the case of DMP/DMP*) of the machines. We call such queries collectively as *subset* queries. Similar to Nath and Venkatesan [2013], we focus only on *static* subset queries, that is, in scenarios where the queries are known to the machines a priori (Nath and Venkatesan [2013] provide a detailed discussion on the hardness of *dynamic* subset queries that also applies to our scenario). Section 7.1 describes how to authenticate dynamic subvector sum queries, Section 7.2 discusses the case of submatrix products, and Section 7.3 covers the setting of subvector dot products.

7.1. Dynamic Subvector Sum Authentication

Let $\mathcal{Q}$ be the set of the clients' queries and $|\mathcal{Q}|$ the cardinality of $\mathcal{Q}$. The result of a subvector sum query $Q_{i_l,i_u} \in \mathcal{Q}$ at τ is defined as

$$Q_{i_l,i_u}\left(\bigcup_{z=1}^{m} \mathcal{X}_z(\tau)\right) = \sum_{z=1}^{m} \mathbf{a}_z[i_l : i_u],$$

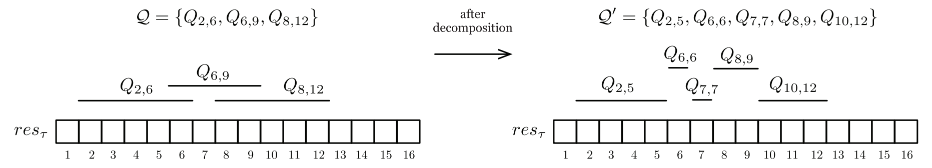
that is, it is computed similar to DVS*, but focusing only on those elements with indices $i_l, i_l+1, \ldots, i_u$ in the machines' vectors (indicated by the notation $(\mathbf{a}_z[i_l : i_u])$).

A naive way to authenticate all queries in $\mathcal{Q}$ is to run $|\mathcal{Q}|$ different instantiations of DVS*, one for each query in $\mathcal{Q}$. However, this would increase all costs at all parties involved by a factor of $|\mathcal{Q}|$. In the following, we first describe a methodology presented in Nath and Venkatesan [2013] for the case of untrusted clients, which can be applied to our setting in combination with DVS*. This method reduces the update cost at the machines from $O(|\mathcal{Q}|)$ to constant. Subsequently, we provide two optimizations of this technique, the first regarding the verification cost and the second concerning the proof generation cost at the server.

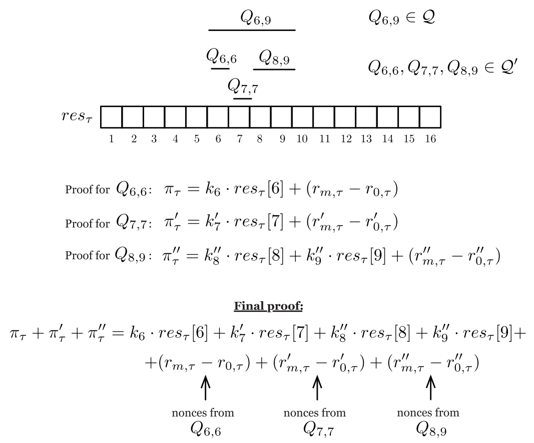
At a preprocessing stage, the technique in Nath and Venkatesan [2013] first creates a new set $\mathcal{Q}'$ from $\mathcal{Q}$ as follows. It retrieves the set $I = \bigcup_{Q_{i_l,i_u} \in \mathcal{Q}}\{(i_l, Q_{i_l,i_u}.qid),$ $(i_u, Q_{i_l,i_u}.qid)\}$, which contains two tuples for every $Q_{i_l,i_u} \in \mathcal{Q}$, where the first (respectively, second) includes the lower (respectively, upper) bound of the subvector range $[i_l, i_u]$ of Q_{i_l,i_u}, along with the unique identifier $Q_{i_l,i_u}.qid$ of Q_{i_l,i_u}. Subsequently, it sorts the tuples $(i, qid) \in I$ in ascending order of i, producing an ordered list L. Finally, in a single scan of L and by checking adjacent elements, it produces a set of disjoint queries $\mathcal{Q}'$ that collectively cover the same elements as the queries in $\mathcal{Q}$. The algorithm is conceptually simple and so we do not further detail it. Instead, we illustrate an example in Figure 15(a), where three overlapping client queries $Q_{2,6}$, $Q_{6,9}$, $Q_{8,12}$ over a 16-element result vector are decomposed into disjoint queries $Q_{2,5}$, $Q_{6,6}$, $Q_{7,7}$, $Q_{8,9}$, $Q_{10,12}$. The cost for producing $\mathcal{Q}'$ is $O(|\mathcal{Q}| \log |\mathcal{Q}|)$, where the size of $\mathcal{Q}'$ is $|\mathcal{Q}'| \leq 2 \cdot |\mathcal{Q}|$.

The system is set in motion and runs $|\mathcal{Q}'|$ different instantiations of DVS*, one for each query in $\mathcal{Q}'$. In order to create the proof for a query $Q \in \mathcal{Q}$, the server identifies the disjoint queries in $\mathcal{Q}'$ that compose Q, and sums up the corresponding proofs from the separate instantiations. This is demonstrated in Figure 15(b) focusing on query $Q_{6,9} \in \mathcal{Q}$, which is comprised of $Q_{6,6}$, $Q_{7,7}$ and $Q_{8,9}$ whose proofs are π_τ, π'_τ and π''_τ, respectively. Note that, since each proof is constructed by a different DVS* instantiation, the keys that scale the result elements, as well as the nonces, are different for each query (i.e., they are query dependent). The client can then verify the result $res_\tau[6 : 9]$ using the aggregate proof and the secret key. The benefit of the preceding approach is that, due to the fact that the queries in $\mathcal{Q}'$ are disjoint, an update always changes an element covered by a single query $Q \in \mathcal{Q}'$ and hence, only a single summary must be updated by the machine (that is, the one that corresponds to the DVS* instantiation for Q). Nevertheless, the drawback is that, since every proof for a query in $\mathcal{Q}'$ is produced from an independent DVS* instantiation, the aggregate proof for a query $Q \in \mathcal{Q}$ contains: (i) as many nonces as the number of queries from $\mathcal{Q}'$ that comprise Q (this is evident in Figure 15(b)), and (ii) keys k that depend on the DVS* instantiations involved in the composition of Q. This suggests that the server must notify the client about which queries from $\mathcal{Q}'$ compose his query. Let n_i be the elements covered by query $Q_i \in \mathcal{Q}$, and N_i the number of queries from $\mathcal{Q}'$ that comprise Q_i. Then, the communication cost for Q_i is $O(N_i)$, whereas the verification cost is $O(n_i + N_i)$.

We can eliminate the N_i term from both the communication and verification cost of the prior approach as follows. First, we mandate that every DVS* instantiation


$\mathcal{Q} = \{Q_{2,6}, Q_{6,9}, Q_{8,12}\}$
after decomposition
$\mathcal{Q}' = \{Q_{2,5}, Q_{6,6}, Q_{7,7}, Q_{8,9}, Q_{10,12}\}$
$Q_{2,6}$ $Q_{6,9}$ $Q_{8,12}$
$Q_{6,6}$ $Q_{8,9}$ $Q_{2,5}$ $Q_{7,7}$ $Q_{10,12}$
res_τ
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

(a) query decomposition: the queries are decomposed into a set of disjoint ranges such that each query can be formed as the union of a set of contiguous ranges

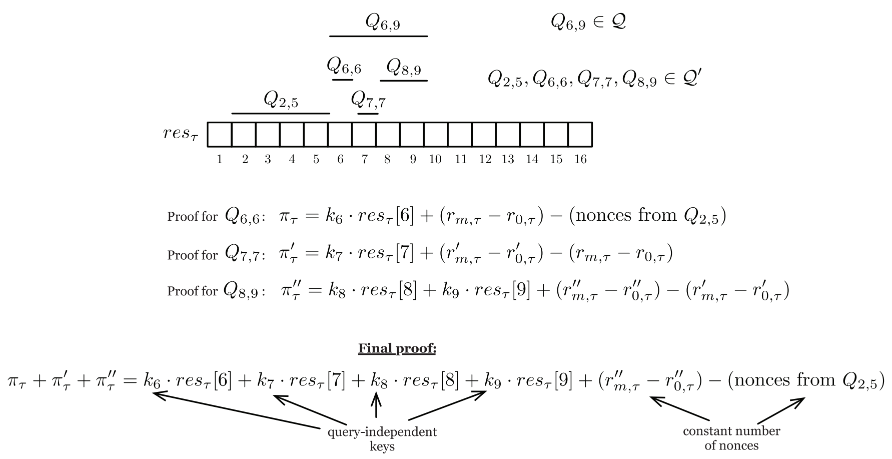

$Q_{6,9}$
$Q_{6,9} \in \mathcal{Q}$
$Q_{6,6}$ $Q_{8,9}$ $Q_{7,7}$
$Q_{6,6}, Q_{7,7}, Q_{8,9} \in \mathcal{Q}'$
res_τ
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Proof for $Q_{6,6}$: $\pi_\tau = k_6 \cdot res_\tau[6] + (r_{m,\tau} - r_{0,\tau})$
Proof for $Q_{7,7}$: $\pi'_\tau = k'_7 \cdot res_\tau[7] + (r'_{m,\tau} - r'_{0,\tau})$
Proof for $Q_{8,9}$: $\pi''_\tau = k''_8 \cdot res_\tau[8] + k''_9 \cdot res_\tau[9] + (r''_{m,\tau} - r''_{0,\tau})$
Final proof:
$\pi_\tau + \pi'_\tau + \pi''_\tau = k_6 \cdot res_\tau[6] + k'_7 \cdot res_\tau[7] + k''_8 \cdot res_\tau[8] + k''_9 \cdot res_\tau[9] +$
$+(r_{m,\tau} - r_{0,\tau}) + (r'_{m,\tau} - r'_{0,\tau}) + (r''_{m,\tau} - r''_{0,\tau})$
nonces from $Q_{6,6}$
nonces from $Q_{7,7}$
nonces from $Q_{8,9}$

(b) proof generation: the proof is formed as a summation of proofs for each range that constitutes the query

Fig. 15. Query and proof with the Nath and Venkatesan [2013] adaptation.

produces the keys k that scale the result elements in a query-independent way. Second, we adapt nonce chaining on the nonces of the different instantiations in the following manner. The machines first sort the queries in $\mathcal{Q}'$ in ascending order of their index ranges. Let the produced ordered list be $L = (Q_1, Q_2, \ldots, Q_\ell)$. Next, they run a different instantiation of DVS* on each Q_i in L, with a slight modification: when executing Sign for Q_{i+1}, the nonces incorporated in the produced signature negate those added in the signature of Q_i. Due to the similarity with the nonce chaining adaptations already discussed for our previous schemes, we do not include the detailed algorithm. Instead, we illustrate the approach with a comprehensive example in Figure 16(a), which continues that of Figure 15(b). Observe that, when $\pi_\tau, \pi'_\tau, \pi''_\tau$ are added together, a constant number of nonces remain in the yielded proof: two related to query $Q_{2,5} \in \mathcal{Q}'$ that precedes the leftmost query $Q_{6,6} \in \mathcal{Q}'$ contained in $Q_{6,9} \in \mathcal{Q}$, and two for the rightmost query $Q_{8,9} \in \mathcal{Q}'$ contained in $Q_{6,9}$. Moreover, the k keys are now query independent. Therefore, the client only needs to know the id of $Q_{2,5}$ and $Q_{8,9}$ (that are used to produce the query-dependent nonces remaining in the final proof) in order to successfully verify the result. These are sent by the server without adding any extra asymptotic cost to the proof size. Consequently, the communication cost at the client for a query $Q_i \in \mathcal{Q}$ covering n_i elements becomes constant, whereas the verification cost becomes $O(n_i)$.

We next analyze the proof generation cost at the server in this approach. The server can produce the proofs for all queries in $\mathcal{Q}$ as follows. It first calculates the proofs for every query in $\mathcal{Q}'$ by summing the m signatures corresponding to each of these queries received by the machines. Next, for every query $Q_i \in \mathcal{Q}$, it identifies those queries from $\mathcal{Q}'$ that comprise it and sums up their proofs. Let N_i be the number of queries from $\mathcal{Q}$ covering a query $Q_i \in \mathcal{Q}$. Then, the total cost is $O(m \cdot |\mathcal{Q}| + \sum_{Q_i \in \mathcal{Q}} N_i)$.



Proof for $Q_{6,6}$: $\pi_\tau = k_6 \cdot res_\tau[6] + (r_{m,\tau} - r_{0,\tau}) - (\text{nonces from } Q_{2,5})$

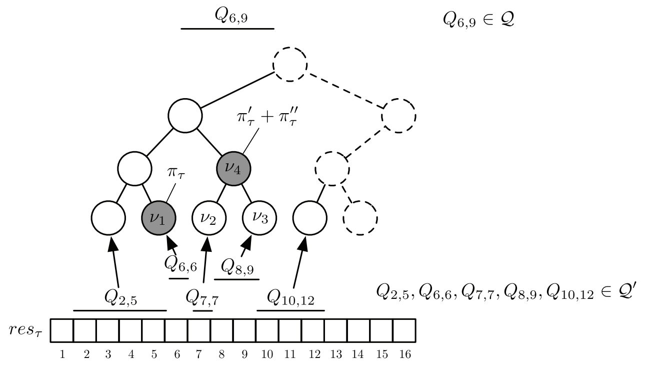
Proof for $Q_{7,7}$: $\pi'_\tau = k_7 \cdot res_\tau[7] + (r'_{m,\tau} - r'_{0,\tau}) - (r_{m,\tau} - r_{0,\tau})$

Proof for $Q_{8,9}$: $\pi''_\tau = k_8 \cdot res_\tau[8] + k_9 \cdot res_\tau[9] + (r''_{m,\tau} - r''_{0,\tau}) - (r'_{m,\tau} - r'_{0,\tau})$

Final proof:

$$\pi_\tau + \pi'_\tau + \pi''_\tau = k_6 \cdot res_\tau[6] + k_7 \cdot res_\tau[7] + k_8 \cdot res_\tau[8] + k_9 \cdot res_\tau[9] + (r''_{m,\tau} - r''_{0,\tau}) - (\text{nonces from } Q_{2,5})$$

(a) applying nonce chaining: the cost to build the proof now depends solely on the number of partitions of the query range, plus a constant



Proof for $Q_{6,6}$: $\pi_\tau = k_6 \cdot res_\tau[6] + (r_{m,\tau} - r_{0,\tau}) - (\text{nonces from } Q_{2,5})$

Proof for $Q_{7,7}$: $\pi'_\tau = k_7 \cdot res_\tau[7] + (r'_{m,\tau} - r'_{0,\tau}) - (r_{m,\tau} - r_{0,\tau})$

Proof for $Q_{8,9}$: $\pi''_\tau = k_8 \cdot res_\tau[8] + k_9 \cdot res_\tau[9] + (r''_{m,\tau} - r''_{0,\tau}) - (r'_{m,\tau} - r'_{0,\tau})$

Final proof:

$$\nu_1 + \nu_4 = \pi_\tau + \pi'_\tau + \pi''_\tau = k_6 \cdot res_\tau[6] + k_7 \cdot res_\tau[7] + k_8 \cdot res_\tau[8] +$$
$$+ k_9 \cdot res_\tau[9] + (r''_{m,\tau} - r''_{0,\tau}) - (\text{nonces from } Q_{2,5})$$

(b) tree structure and final proof generation: partial combination of results within the tree reduces the cost to proportional to the tree height

Fig. 16. Our optimizations for subvector sum authentication.

Our second optimization regards improving this cost to $O(m \cdot |\mathcal{Q}| + \sum_{Q_i \in \mathcal{Q}} \log N_i)$. The server achieves this by constructing a binary tree over the proofs of the queries in $\mathcal{Q}'$, similar to that shown in Figure 16(b). In particular, the server first sorts the proofs of the queries in $\mathcal{Q}'$ in ascending order of the vector indices to which the queries correspond. Then, it associates the proof of every query in $\mathcal{Q}'$ with a leaf node. Subsequently, it constructs the tree in a bottom-up manner storing in each internal node the sum of the values contained in its children. In our figure, $\nu_1 = \pi_\tau$ corresponds to $Q_{6,6}$, whereas $\nu_4 = \pi'_\tau + \pi''_\tau$ is the sum of the proofs of $Q_{7,7}$, $Q_{8,9}$. Observe that every node is associated with an index range in the result vector, for instance, ν_4 corresponds to range $[7, 9]$.

The tree can be built in $O(|\mathcal{Q}'|)$ time. Given a query $Q_i \in \mathcal{Q}$, the server identifies the values stored in the roots of those maximal subtrees that exactly cover the index range of Q_i. For instance, values ν_1 and ν_4 (highlighted in grey) are the values in the roots of the maximal subtrees covering the range $[6, 9]$ of $Q_{6,9} \in \mathcal{Q}$. Eventually, it sums these values to produce the final proof. In our running example, observe that $\nu_1 + \nu_4$ is equal to $\pi_\tau + \pi'_\tau + \pi''_\tau$, which constitutes the final proof for $Q_{6,9}$. Since the number of maximal subtrees is logarithmic in the number of the leaves covering the index range of Q_i, the cost for computing the proof of Q_i is $\log N_i$. As a result, the total proof generation cost at the server for all the queries in $\mathcal{Q}$ becomes $O(m \cdot |\mathcal{Q}| + \sum_{Q_i \in \mathcal{Q}} \log N_i)$. This expression is $O(|Q|(m + \log n))$, since the height of the binary tree is $\log n$, which bounds the cost of $\log N_i$.

The correctness of our nonce chaining and tree structure modification is easily verifiable from Figure 16. Moreover, the security of the nonce chaining adaptation can be proven in a similar vein to our previous constructions, whereas the introduction of the tree structure does not affect security at all (since it does not alter the view of the adversary). Consequently, we omit the formal proofs of correctness and security.

7.2. Dynamic Submatrix Product Authentication

Let $\mathcal{Q}$ be the set of the clients' queries and $|\mathcal{Q}|$ the cardinality of $\mathcal{Q}$. The result of a submatrix product query $Q_{i_l,i_u,j_l,j_u} \in \mathcal{Q}$ at τ is defined as

$$Q_{i_l,i_u,j_l,j_u}(\mathcal{X}_a(\tau) \cup \mathcal{X}_b(\tau)) = \mathbf{AB}[i_l : i_u, j_l : j_u],$$

that is, it is computed similar to DMP*, but focusing only on those elements of the matrix product with indices $[i_l, j_l], [i_l + 1, j_l] \ldots, [i_u, j_u]$.

In order to authenticate this query, we could extend the methodology from Nath and Venkatesan [2013] in a similar manner to that explained in Section 7.1. More specifically, we could view the queries in $\mathcal{Q}$ as two-dimensional ranges in the domain $[n_a] \times [n_b]$, and decompose them into a set $\mathcal{Q}'$ of disjoint rectangular queries that exactly cover $\mathcal{Q}$. Nevertheless, observe that, in the worst case where all queries from $\mathcal{Q}$ overlap with each other, it holds that $|\mathcal{Q}'| = O(|\mathcal{Q}|^2)$. Executing a different DMP* instantiation for every query in $\mathcal{Q}'$ would lead to a prohibitive cost at the machines for producing and sending the individual signatures. Note that our two optimizations in the case of subvector sums did not target at the machine's overheads and hence their adaptation here cannot alleviate these costs. Consequently, in the case of submatrix products, we advocate the use of different instantiations of DMP* for every query in $\mathcal{Q}$ (and not for every query in $\mathcal{Q}'$), which may only impact the update cost at the machines in case numerous queries overlap at the same elements.

7.3. Dynamic Subvector Dot Product Authentication

Let $\mathcal{Q}$ be the set of the clients' queries and $|\mathcal{Q}|$ the cardinality of $\mathcal{Q}$. The result of a subvector dot product query $Q_{i_l,i_u} \in \mathcal{Q}$ at τ is defined as

$$Q_{i_l,i_u}(\mathcal{X}_a(\tau) \cup \mathcal{X}_b(\tau)) = \mathbf{a}[i_l : i_u] \cdot \mathbf{b}[i_l : i_u],$$

that is, it is computed similar to DDP, but focusing only on those elements of the machines' vectors with indices $i_l, i_l + 1, \ldots, i_u$.

We follow a similar approach to that described in Section 7.1 for subvector sum. We decompose $\mathcal{Q}$ into a set $\mathcal{Q}'$ of disjoint queries that exactly cover $\mathcal{Q}$. The machines run a different instantiation of DDP for each query in $\mathcal{Q}'$. However, it is vital to note that the keys which scale the elements in the summaries are constructed in the same manner as in DDP, that is, in M_a (M_b) element $\mathbf{a}[i]$ (respectively, $\mathbf{b}[i]$) is always scaled with key k^i (respectively, k^{n-i+1}), regardless of which query in $\mathcal{Q}'$ covers this element.

We make two observations focusing on ℓ queries $Q_1, \ldots, Q_\ell \in \mathcal{Q}'$ that exactly cover a client's query $Q \in \mathcal{Q}$: (i) If the server creates proof $\pi_{\tau,i}$ for query $Q_i \in \mathcal{Q}'$ in the same manner as in DDP (after collecting the appropriate signatures from the machines), then the product $\prod_{i=1}^{\ell} \pi_{\tau,i}$ will give a proof of the form $g^{k^{n+1} \cdot \sum_{j=i_l}^{i_u} res_\tau[j] + (\text{nonces})}$ for the subvector result $res_\tau[i_l : i_u] = \mathbf{a}[i_l : i_u] \cdot \mathbf{b}[i_l : i_u]$ at τ corresponding to query Q. Consequently, we can create a tree structure on the proofs of the queries in $\mathcal{Q}'$ as in Figure 16(b), with the difference that an internal node now stores the product of the values stored in its children. (ii) If the nonces are produced using nonce chaining in the same vein as for the subvector sum, then the number of nonces in the proof of Q described before will be constant. Therefore, subvector dot product can make use of our two optimizations described in Section 7.1, enjoying their benefits, namely: (i) the constant update cost at the machines, (ii) the reduced verification time at the client, and (iii) the faster proof generation overhead at the server.

8. APPLICATIONS

In this section, we discuss some common queries in stream outsourcing and explain how the constructions we have provided can address them. We stress, though, that the applicability of our schemes is not limited to these cases; we are confident that our fundamental tools can capture a much wider set of applications.

8.1. Group-by Queries

The class of `group-by,sum` aggregation queries are at the heart of many outsourced computation scenarios and have been the sole motivation for much of the prior work on stream authentication. The setting is that a large number of tuples are observed as a stream. These tuples may correspond, for example, to activity on a network, updates to a large database table, or events in an event processing system. The requirement is for the server to collate the stream tuples into groups and report the sum for each group. We typically consider cases where the number of active groups (those with a nonzero sum) is substantially large, so that the data owner benefits from enlisting the server to perform the aggregation.

This problem is solved directly by the dynamic vector sum authentication protocol, DVS (and its optimized version DVS*), applied to a single vector. Each stream tuple is translated into an update to the vector. The entries of the vector give the aggregate associated with each corresponding group. The approach naturally holds for the distributed setting, where updates might be spread across multiple streams. In this case, the object of the authentication is the vector given by the sum of the vectors derived from each of the streams. DVS and DVS* capture this scenario due to their homomorphic property that allows the client to verify a sum of vectors by checking a single proof (produced by the server) that combines all individual vector signatures together.

Furthermore, sometimes a client may require statistics over a period of time. For example, in a stock market application, the client may wish to learn the aggregate of a stock (or a group of stocks) for the last w timestamps (where a timestamp may correspond to a minute, a day, etc.), for instance, for computing moving averages or other statistics over time. Our W-DVS scheme can be used to authenticate such sliding window queries.

Finally, in several applications a client may be interested in only a subset of groups. For instance, in an electronic bookstore application, customer transactions form the streams and the client may wish to learn statistics only about certain purchases, such as those concerning only math books, novels, etc. To handle such cases, we can sort the book ids by category and directly apply our adaptation of DVS/DVS* to the subset queries scenario.

8.2. Join Queries

Beyond simple grouping and aggregation, many important outsourced queries involve the computation of a `join` query on relations. In traditional data stream management systems, join queries are regarded as particularly challenging, with prior work focusing on approximate results [Das et al. 2003; Viglas et al. 2003]. Hence, join queries are a prime candidate for outsourcing. We explain how to authenticate join results in our setting, focusing on the common case of equi-join.

We consider two machines M_a and M_b. At every epoch, machine M_a (M_b) generates data that dynamically update a relation R (respectively, S) which is maintained at the server. An update t at M_a has the form $(sign, t_R)$, where t_R corresponds to a relation record and $sign$ is either "+" or "−"; the former indicates that t_R is added to R, whereas the latter signifies that t_R is deleted from R. The case for machine M_b is similar.

Given a long-running equi-join query

```
SELECT * FROM R, S
WHERE R.x = S.x
```

the server returns to the client the result at every epoch, which is a multiset of tuples $(t_R, t_R.x, t_S)$ where t_R is a tuple from (the current version of) R, t_S is a tuple from (the current version of) S, and $t_R.x = t_S.x$ is their common value on the join attribute x. We assume that the domain of x is $[n]$, where n is an integer. We next explain that we can adapt our DDP construction with slight modifications in order to authenticate the aforesaid equi-join query. We call the resulting construction as DEJ.

We first explain the main idea behind the scheme. Sending their data streams to the server, machines M_a and M_b dynamically update two n-element vectors $\mathbf{a}$ and $\mathbf{b}$, respectively. At the beginning of the system, these vectors are initialized with zeros. When a new tuple $(+, t_R)$ arrives at M_a, then $H(t_R)$ is added to the current value of $\mathbf{a}[t_R.x]$, where H is a cryptographic hash function (e.g., SHA-1 [Menezes et al. 1996]). On the other hand, if $(-, t_R)$ arrives at M_a, then $H(t_R)$ is subtracted from the current value of $\mathbf{a}[t_R.x]$. In other words, at the end of any epoch, element $\mathbf{a}[i]$ contains the sum of the hashes of the records in the current version of R, whose join attribute value is i. The case for M_b and $\mathbf{b}$ is similar. Then, the dot product $\mathbf{a} \cdot \mathbf{b}$ encompasses cryptographic information only about those tuples that participate in the join result at the current epoch. This is because, if there are no tuples joining on attribute value j, then at least one of $\mathbf{a}[j]$, $\mathbf{b}[j]$ must be equal to zero. Therefore, authenticating this dot product result (through our DDP functionality) allows us to authenticate the desired equi-join result.

Figure 17 contains the pseudocode of DEJ (focusing on the case of machine M_a). We properly modify procedure Sign, such that M_a maintains summary $\mathcal{S}_a$ on the vector $\mathbf{a}$ described earlier. Note that, for machine M_b, the summary is updated as $\mathcal{S}_b = \mathcal{S}_b \pm k^{n-t_S.x+1} \cdot H(t_S)$. Contrary to the case of DDP where the client receives just a single value as the dot product result, here it receives those tuples that participate in the join result. As such, Verify first needs to use these tuples to compute the elements of $\mathbf{a}$ and $\mathbf{b}$ that participate in $\mathbf{a} \cdot \mathbf{b}$. In other words, for every distinct join attribute value x, the user computes $\mathbf{a}[x]$ ($\mathbf{b}[x]$) as the summation of the hashes of the records from R (respectively, S) included in res_τ. Finally, it computes the sum of all $\mathbf{a}[x] \cdot \mathbf{b}[x]$ scaled with key k^{n+1}, injects the appropriate nonces, and verifies the yielded value against π_τ.

Correctness and security. We prove the correctness and security of DEJ in the two theorems that follow.

```
// Algorithms KeyGen, Sign, and Combine are the same as in the DDP construction (Figure 4)

Update($a, sk, \mathcal{S}_a, t$)
1. Parse $t$ as $(sign, t_R)$
2. If $sign = `+'$, then $S_a = S_a + k^{t_R.x} \cdot H(t_R)$
3. Else $S_a = S_a - k^{t_R.x} \cdot H(t_R)$
4. Output $\mathcal{S}_a$

Verify($sk, \pi_\tau, res_\tau, \tau$)
1. Parse $sk$ as $k$, and $res_\tau$ as a multiset of tuples of the form $(t_R, t_R.x, t_S)$
2. $r_{a,\tau} = F_k(\text{``machine''}\|a\|\text{``epoch''}\|\tau\|\text{``}r\text{''})$
3. $\rho_{a,\tau} = F_k(\text{``machine''}\|a\|\text{``epoch''}\|\tau\|\text{``}\rho\text{''})$
4. $r_{b,\tau} = F_k(\text{``machine''}\|b\|\text{``epoch''}\|\tau\|\text{``}r\text{''})$
5. $\rho_{b,\tau} = F_k(\text{``machine''}\|b\|\text{``epoch''}\|\tau\|\text{``}\rho\text{''})$
6. Initialize $\pi = g^{(r_{a,\tau} \cdot r_{b,\tau} - \rho_{a,\tau} - \rho_{b,\tau})}$
7. Compute $\pi = \pi \cdot g^{k^{n+1} \cdot \sum_x \left(\left(\sum_{(t_R, t_R.x, t_S) \in res_\tau} H(t_R)\right) \cdot \left(\sum_{(t_R, t_R.x, t_S) \in res_\tau} H(t_S)\right)\right)}$
8. If $\pi = \pi_\tau$ output Yes, otherwise No
```

Fig. 17. The DEJ construction.

THEOREM 8.1. DEJ *is correct.*

PROOF. The correctness of DEJ stems from that of the underlying DDP scheme. Recall that, in DEJ, at the end of every epoch τ it holds that $\mathbf{a}[i] = \sum_{t_R \in R \wedge t_R.x=i} H(t_R)$ and $\mathbf{b}[i] = \sum_{t_S \in S \wedge t_S.x=i} H(t_S)$, where R and S correspond to the instances of the two queried relations at τ. Since Sign and Combine in DEJ are identical to those in DDP, the server sends to the client proof $\pi_\tau = g^{k^{n+1} \cdot (\mathbf{a} \cdot \mathbf{b}) + r_{a,\tau} \cdot r_{b,\tau} - \rho_{a,\tau} - \rho_{b,\tau}}$. Observe that Verify outputs **Yes**, if $\sum_x \left(\sum_{(t_R, t_R.x, t_S) \in res_\tau} H(t_R)\right) \cdot \left(\sum_{(t_R, t_R.x, t_S) \in res_\tau} H(t_S)\right) = \mathbf{a} \cdot \mathbf{b}$, which holds when res_τ is the equi-join result. □

THEOREM 8.2. *If F is a PRF, then* DEJ *is secure under the n-DHE assumption in the random oracle model.*

The proof is in Appendix A.7.

Performance. Algorithms KeyGen, Sign, and Combine in DEJ are identical to those in DDP and hence retain their costs. The slight modification in the Update procedure negligibly affects its cost as compared to DDP (it requires one extra hash operation). Finally, let $N_R = |\{t_R | \exists (t_R, t_R.x, t_S) \in res_\tau\}|$ and $N_S = |\{t_S | \exists (t_R, t_R.x, t_S) \in res_\tau\}|$, that is, N_R (N_S) is the number of tuples from R (respectively, S) participating in the join result. Then, the cost of Verify in DEJ is dominated by $O(N_R + N_S)$ hash operations and a constant number of modular exponentiations.

Other Join Variations. In the aforementioned problem, the goal was to authenticate the tuples produced by an equi-join query on relations R, S on attribute x. Assume that $R.y$ and $S.z$ are attributes of relations R and S, respectively. If, instead of the actual tuples, we are interested in authenticating the joint frequency distribution of each $(R.y, S.z)$ pair in the join result, then this authentication can be achieved by a direct application of our DMP protocol. In this case, the machine containing relation R (S) builds a two-dimensional matrix where each element of the matrix corresponds to the joint frequency of occurrences of each $(R.y, R.x)$ (($S.x$, $S.z$)) pair of values in R (S). It is easy to see that the product of these two matrices provides the desired result in this application.

Another interesting query is computing the size of the equi-join result. This is given by a direct application of DDP: if we treat every tuple t with join value $t.x$ as an update of the form $(sign, t.x, 1)$, then vectors $\mathbf{a}, \mathbf{b}$ will hold the frequencies of the relations on each join value. Therefore the equi-join size is exactly $\mathbf{a} \cdot \mathbf{b}$.

Finally, observe that we can handle join queries with a range selection on the join attribute, using our DDP implementation for subset queries. This is because a range selection on the join attribute essentially constrains the query to a subvector of $\mathbf{a}$ and $\mathbf{b}$, which is directly supported by the solution we presented in Section 7.3.

8.3. In-Network Aggregation

In-network aggregation is a popular paradigm employed typically in sensor networks, which reduces the energy expenditure in routing raw data from the motes to a remote client [Madden et al. 2002]. Consider a set of sensors organized (without loss of generality) into a tree-structured network. Also assume that a client communicates only with the root sensor (sink) and wishes to perform some aggregation task (e.g., `sum` or `count`) on the readings of the sensors. Transmitting the raw data to the client inflicts considerable burden on the nodes positioned close to the sink, as they have to forward a considerable number of messages from nodes lower in the tree. In-network aggregation mandates that internal nodes perform the aggregation task on the data received from their children and forward only a small result, thus achieving significant battery savings.

In our setting, only the leaf sensors belong to the owner, whereas the internal tree infrastructure is outsourced to an untrusted third party [Garofalakis et al. 2007; Papadopoulos et al. 2011; Nath et al. 2009]. The goal is to allow the client to authenticate the aggregation result on the union of the leaf readings received from the sink. Our DVS (or DVS*) construction applies to this scenario as well. Its KeyGen, Update, Sign, and Verify routines remain the same in this case; the main changes occur in the Combine algorithm executed by the server. Here, each server in the network executes Combine on the inputs received from its children, and forwards the output to its parent in the routing tree. Notice that the client eventually receives a proof from the sink that is equal to the summation of the leaf sensor signatures. This is exactly what Combine would output in case a single server collected all the sensor signatures. Our scheme is lightweight for all parties involved and hence is ideal for resource-constrained sensor networks.

Subset and sliding window queries naturally arise in the aforesaid setting as well. For instance, in case the sensors transmit multiple sensor readings, a client may wish aggregate information only about a subset of these readings. Moreover, the client may request statistics (e.g., about temperature) over the last w epochs. Therefore, the subset and sliding window adaptations of our DVS/DVS* schemes have immediate application in these scenarios.

8.4. Similarity Measures

It is increasingly common to deal with objects represented by a (potentially) very large number of features in a high-dimensional vector space. In machine learning and other modeling applications, a single object (such as a user of a Web search engine) may be represented by a vector that has millions or billions of components. Similarity measures are vital in such settings. For instance, clustering of objects often entails distance (i.e., dissimilarity) computation between feature vectors. Another example involves determining the correlation of items (i.e., market stocks, retail products, etc.) whose information (i.e., shares values, sales volume) is dispersed across different server machines. Correlation is also based on similarity.

Similarity between vectors is typically measured by an appropriate similarity or dissimilarity measure, such as the cosine similarity and Euclidean distance, respectively. The cosine similarity between vectors $\mathbf{a}, \mathbf{b}$ is computed as $\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|}$, where $\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$ is the L_2 norm of $\mathbf{a}$. The Euclidean distance between $\mathbf{a}$ and $\mathbf{b}$ is equal to $\|\mathbf{a} - \mathbf{b}\| = \sqrt{\|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 - 2\mathbf{a} \cdot \mathbf{b}}$.

We can authenticate such measures by engaging the DDP construction, since both cosine similarity and Euclidean distance depend on inner product computation. The case of $\mathbf{a} \cdot \mathbf{b}$ is carried out by direct application of DDP. For $\mathbf{a} \cdot \mathbf{a}$ and $\mathbf{b} \cdot \mathbf{b}$, M_a and M_b must apply two separate instances (i.e., with different keys) of DDP on the same vector $\mathbf{a}$ and $\mathbf{b}$, respectively. The modifications in Combine and Verify are straightforward and thus omitted. In addition, similarity may be defined over a subset of features, or over a period of time (e.g., stock similarity). Hence, the subset query DDP adaptation and W-DDP scheme are applicable in these cases.

8.5. Event Co-Occurrence

Event monitoring applications operate on massive streams in order to find patterns or correlations between certain events [Demers et al. 2007]. These include supply chain management of RFID tagged products, stock trading, monitoring of machines for malfunctions, environmental sensing for surveillance of establishments, and more. An important class of queries in this scenario is finding *co-occurrence* of events. We provide a simple example. Let $\mathbf{A}$ (respectively, $\mathbf{B}$) be an $n \times 1$ ($1 \times n$) matrix representing a set of n events occurring at machine M_a (M_b). A cell value is 1 if the event occurs during the latest (or at a specific) epoch, and 0 otherwise. Then, $\mathbf{AB}$ is an $n \times n$ matrix where cell $\mathbf{AB}[i][j]$ is 1 if event i co-occurs with event j at the latest (or at a specific) epoch. The result matrix can help in determining event correlations. The preceding can be generalized to matrices with arbitrary dimensions. It is apparent that our DMP construction is directly applicable for authenticating such queries.

Moreover, event co-occurrence is meaningful when observed under a sliding window. For instance, suppose the rows in matrix $\mathbf{A}$ correspond to events that occur at machine M_a. A new column appended at τ to $\mathbf{A}$ indicates which events occurred at τ at M_a. Similarly, consider that the columns in matrix $\mathbf{B}$ refer to events that occur at machine M_b. A new row appended at τ to $\mathbf{B}$ indicates which events happened at τ at M_b. Then, the matrix product $\mathbf{AB}$ over a sliding window of size w will contain in cell $[i, j]$ the number of times event i co-occurred with event j in the last w epochs. Clearly, such queries can be authenticated by our W-DMP scheme.

9. EXPERIMENTS

In this section we experimentally evaluate our protocols, namely DVS, DMP, DDP, DVS*, DMP*, W-DVS, W-DMP, and W-DDP. We also compare DVS with PIRS (specifically, PIRS-1 [Yi et al. 2009]), which is the only scheme that addresses our trusted client setting in the context of `group-by,sum` queries. However, we stress that PIRS is not a direct competitor, as it assumes the client is the owner itself and has a weaker security model. We slightly adapt PIRS so that the machines send to the client their summaries via the server, after authenticating them using another authentication scheme. On the other hand, as we are the first to address authentication of dot and matrix products as well as of sliding window queries in the trusted client setting, the rest of our schemes have no competitors.

The remainder of this section is outlined as follows. Section 9.1 includes information about the implementation of our constructions, along with an evaluation of the primitive costs. Section 9.2 assesses the performance of our basic constructions. Section 9.3 evaluates our optimized constructions DVS* and DMP*, while Section 9.4 focuses on sliding window queries. Finally, Section 9.5 summarizes our experimental results.

Table II. Primitive Costs

Description	Cost
Modular addition in $\mathbb{Z}_p$ ($\lvert p\rvert = 10$ / 20 bytes)	0.15 μs / 0.18 μs
Modular multiplication in $\mathbb{Z}_p$ ($\lvert p\rvert = 10$ / 20 bytes)	0.19 μs / 0.28 μs
Modular exponentiation in $\mathbb{Z}_p$ ($\lvert p\rvert = 10$ / 20 bytes)	4.8 μs / 7.4 μs
Time to derive the generator of $\mathbb{Z}_q$ ($\lvert q\rvert = 64$ bytes)	2.7 *sec*
Modular multiplication in $\mathbb{Z}_q$ ($\lvert q\rvert = 64$ bytes)	0.56 μs
Modular exponentiation in $\mathbb{Z}_q$ ($\lvert q\rvert = 64$ bytes)	55.6 μs
HMAC computation (with SHA-1)	3.53 μs

9.1. Implementation

We implemented all protocols in C on a 2.66 GHz Intel Core i7 with 4GB of RAM, running MAC OS X. We used the GMP[3] and OpenSSL[4] libraries for implementing the cryptographic operations involved. We utilized HMAC with SHA-1 [Menezes et al. 1996] for the F function, which produces 20-byte outputs. We employed HMAC with SHA-1[5] also as the message authentication scheme in PIRS for authenticating the summaries to the clients.

An important discussion concerns the selection of the size of the prime p that defines the $\mathbb{Z}_p$ domain (i.e., the value for security parameter s). This can be as small as 10 bytes for safeguarding against guessing attacks on the keys. However, in DDP this must be at least 20 bytes. The reason is that DDP relies on the discrete logarithm problem. The well-known Pollard rho algorithm takes $O(\sqrt{p})$ steps to find a logarithm in $\mathbb{Z}_p$ [Menezes et al. 1996], suggesting the size of p should be twice as long as the one that protects against simple guessing.

Furthermore, we computed the generator of the group used in DDP employing the implementation techniques included in Menezes et al. [1996]. Specifically, the element of order p that generates our group of concern $\mathbb{G}$ is selected from $\mathbb{Z}_q$, where q is a 64-byte prime of the form $q = 2\ell p + 1$ [Menezes et al. 1996]. All computations in $\mathbb{G}$ are modulo q. Table II includes the average cost (over 10,000 runs) of each primitive operation entailed in the implemented protocols.

9.2. Basic Constructions

Evaluation of DVS *vs. PIRS*. We compared DVS with PIRS using the World Cup Dataset.[6] The latter contains Web server logs from the 1998 Soccer World Cup. Each log entry consists of a client id, the id of the requested URL, the size of the response, etc. We used the first 2 million tuples from the log of day 50. From each tuple in this set, we produced a tuple (j, v), where j is a client id and v is the size of the response. We then focused on a `group-by`,`sum` query that returns a vector where the j^{th} element corresponds to a unique client j and the value of the j^{th} element is the sum of response sizes of all requests issued by client j.

Table III illustrates the various costs we evaluated during our experiment, assuming a single stream generated by a single machine. We decomposed PIRS into algorithms of the form Update, Sign, and Verify (Combine has no cost in the single-machine setting in both schemes and thus is omitted). The average number of nonzero elements in

[3]http://gmplib.org/.
[4]http://www.openssl.com/.
[5]Note that, although it is well known that SHA-1 is prone to collision attacks, these attacks do not have any effect when SHA-1 is used in HMAC [Bellare 2006], that is, HMAC does not suffer the same weaknesses that have been found in the underlying hash functions. The security of HMAC depends on the key size, which is sufficient in our implementation (10 bytes) to withstand brute-force attacks.
[6]http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

Table III. Comparison of DVS with PIRS (single machine)

Evaluated Cost	DVS	PIRS
CPU time for Update	5.3 μs	2.3 μs
CPU time for Sign	4.8 μs	4.7 μs
CPU time for Verify	48.9 ms	19.1 ms
Summary size	10 bytes	10 bytes
Proof size	10 bytes	30 bytes

Table IV. Comparison of DVS with PIRS ($m = 100$ machines)

Evaluated Cost	DVS	PIRS
CPU time for Combine	10.1 μs	-
CPU time for Verify	50.16 ms	19.69 ms
Proof size	10 bytes	3000 bytes

the result vector (that affects the CPU time in Verify) was around 12,000. PIRS and DVS have comparable CPU overheads for Sign. However, PIRS outperforms DVS for Update and Verify because, contrary to DVS, it does not involve HMAC invocations. Recall, though, that this performance advantage of PIRS comes at the expense of a weaker security model. Moreover, observe that the CPU times for DVS are on the order of a few microseconds at the owner ($5.3\mu s$ for Update and $4.8\mu s$ for Sign), and a few milliseconds at the client (48.9ms for Verify). The summary and proof size is negligible in DVS (10 bytes), while summary size in PIRS is the same, but its proof size is 20 bytes longer due to the additional HMAC that authenticates the summary.

Table IV depicts the costs in the scenario where we repeat the previous experiment, but now the tuples are generated by $m = 100$ machines. The Update and Sign costs are unaffected by m and hence omitted. In PIRS, there is no Combine cost, since the server simply forwards m summaries and HMACs to the client. This considerably increases the total proof size to 3,000 bytes. On the other hand, in DVS, the server combines the signatures of all the machines into a single one, always maintaining the communication cost of 10 bytes. This comes with a very small overhead for the server due to Combine ($10.1\mu s$). The cost of Verify increases by the m extra hash computations in both DVS and PIRS. However, note that the overall cost is rather dominated by the operations imposed by the n vector elements and therefore the overhead is very similar to the case of a single machine in both DVS and PIRS.

Evaluation of DMP. We consider the costs for matrix multiplication between two $n \times n$ matrices. Here, we generate synthetic data by randomly filling entries—note that the data itself does not affect the performance of the DMP construction, as the steps taken are largely data independent. Table V shows the time costs of each of the operations as n varies. The Update step is similar in all cases ($\sim 6\mu s$), as it does not depend on n. The Sign operation scales linearly with n (proportionally to the square root of the input size), exactly as predicted by our analysis. Even for large matrices with hundreds of thousands of entries, this cost is on the order of a few milliseconds; extrapolating to billion-entry matrices, the cost will remain below a second. Combine scales similarly, proportional to the size of the summary. Only Verify is more expensive due to the cost of reading the full $n \times n$ result, performing modular multiplications for each entry, and invoking $O(n)$ HMAC calls. Yet this too is way below a second even for our largest example.

Evaluation of DDP. We give our results for DDP in Table VI. Here, we also generate synthetic vectors of differing sizes. Observe there is a nontrivial setup cost for this protocol, which stems from determining a generator for $\mathbb{G}$ and computing the

Table V. Scalability of DMP with n ($n_a = n_b = n$)

Evaluated Cost	$n = 5$	$n = 50$	$n = 500$
CPU time for Update	5.5 μs	5.4 μs	6.0 μs
CPU time for Sign	58.4 μs	567 μs	5.7 *ms*
CPU time for Combine	3.0 μs	24.3 μs	263 μs
CPU time for Verify	0.13 *ms*	2.13 *ms*	78.3 *ms*

Table VI. Scalability of DDP with n

Evaluated Cost	n = 100	n = 1000	n = 10000
CPU time for KeyGen	2.8 *sec*	2.9 *sec*	3.9 *sec*
CPU time for Update	2.58 μs	3.38 μs	4.3 μs
CPU time for Sign	14.5 μs	13.95 μs	14.6 μs
CPU time for Combine	2.43 *ms*	30.75 *ms*	538 *ms*
CPU time for Verify	129 μs	143 μs	160 μs

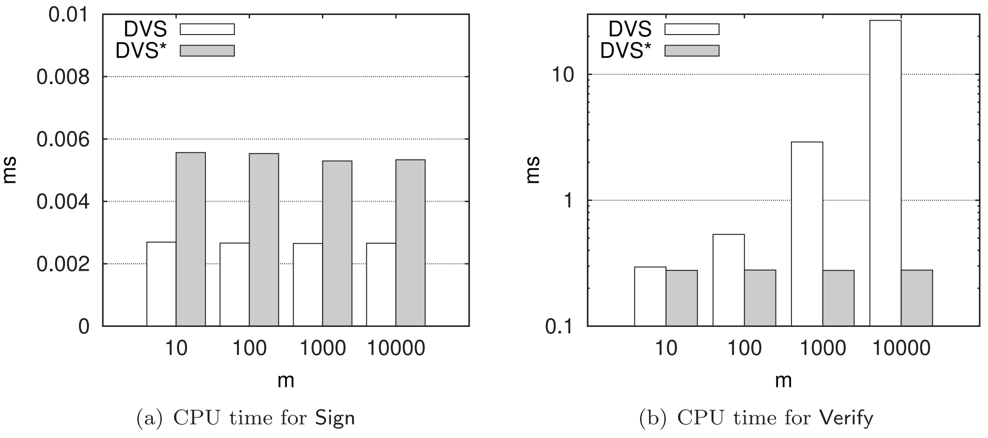
exponentiated values in *pub*. However, most of the work is in finding a suitable generator, although this truly is a one-time operation. The cost varies little with the vector size n. As before, Update does not depend on n, and in this case neither does Sign. Therefore, the two overheads are relatively unaffected by n. Our cost for Combine grows linearly with n, as predicted by our performance analysis, and remains below one second even in our worst-case experiment ($n = 10{,}000$). The cost for Verify is quite low since it requires only a constant amount of light work for checking the proof.

9.3. Optimized Constructions

We compare DVS* against DVS, and assess DMP* versus DMP, in order to show the effectiveness of our nonce chaining technique. We experimented with synthetic data since, as we mentioned before, our methods are not significantly affected by the data distribution.

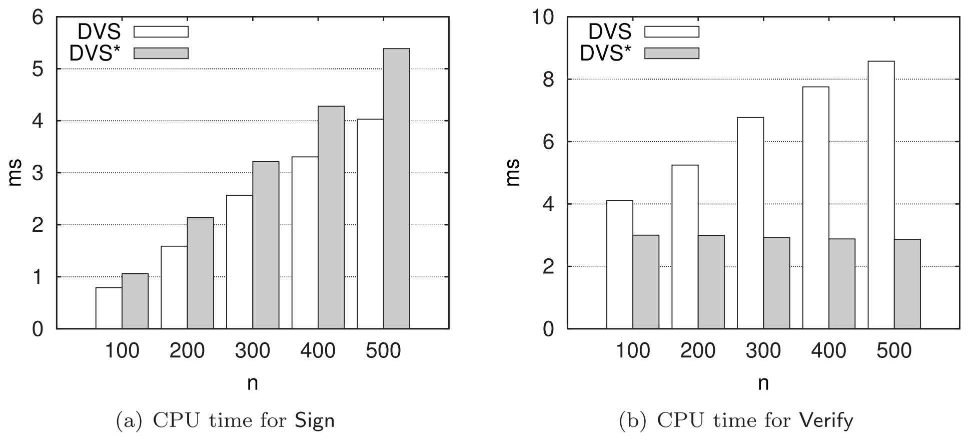
Evaluation of DVS* *vs.* DVS. Recall that the two methods differ only in algorithms Sign and Verify. As such, in Figure 18 we plot only the running times of these two processes while varying the number of machines m and setting the vector size n to 100. Figure 18(a) demonstrates the running time of Sign (in ms) as a function of m. This cost is independent of w, since Sign involves a constant number of nonce generations and additions. The cost in DVS* is twice that of DVS, as it entails one extra nonce generation and addition as compared to its counterpart for enforcing the nonce chaining technique. Observe, though, that the overhead in DVS* ($\sim 6\mu s$) is lightweight and comparable to that in DVS ($\sim 3\mu s$).

The main gain of DVS* versus DVS is in the verification cost that is realized through the nonce chaining technique. Figure 18(b) illustrates the CPU time for Verify when varying m. Although this cost in DVS increases linearly with m (since the client must reconstruct the m nonces contributed by the machines in the final proof), the time remains constant in DVS*. This is because nonce chaining eliminates the nonces incorporated by the machines, leaving exactly two nonces to be reconstructed by the client. The verification time in DVS* always remains $\sim$0.28ms, whereas in DVS it ranges from 0.29ms to 27ms, rendering our optimization faster by up to two orders of magnitude. As an additional remark, observe that the benefit of nonce chaining is more pronounced when the number of machines m becomes larger than the vector size n, since the client must also evaluate the PRF function and perform modular multiplications/additions for every element in the result vector.



(a) CPU time for Sign (b) CPU time for Verify

Fig. 18. DVS* vs. DVS when varying m ($n = 100$).



(a) CPU time for Sign (b) CPU time for Verify

Fig. 19. DMP* vs. DMP when varying n ($n_a = n_b = 100$).

For completeness, we also include a brief discussion on the common costs of the two methods. The overhead of Update is independent of m and thus always remains $\sim 3.5\mu s$. On the other hand, the time of Combine increases linearly with m, since the server must sum all the signatures from all the machines. Nevertheless, this cost ranges from $0.6\mu s$ ($m = 10$) to $663\mu s$ ($m = 10{,}000$), which is lightweight even for a large number of machines. Finally, the time for KeyGen is negligible.

Evaluation of DMP* *vs.* DMP. Similar to DVS*/DVS, the two matrix product schemes differ only in Sign and Verify. Figure 19 illustrates the costs of these algorithms when authenticating the product between an $n_a \times n$ matrix and an $n \times n_b$ matrix. In particular, Figure 19(a) assesses the cost (in ms) of Sign in DMP* and DMP when varying n and setting $n_a = n_b = 100$. This overhead increases linearly with n, since machine M_a (respectively, M_b) must produce one signature for every column (respectively, row). The time ranges between 0.79ms and 4ms in DMP, and between 1.06ms and 5.4ms in DMP*. Observe that the cost is slightly higher (by 20–35%) in DMP* than in DMP, due to the extra nonce generations mandated by nonce chaining. However, the overhead of DMP* is still lightweight.

Table VII. Performance of W-DVS when Varying w ($m = 100$)

Evaluated Cost	w = 10	w = 100	w = 1000	w = 10000
CPU time for Update	3.57 μs	3.54 μs	3.60 μs	3.44 μs
CPU time for Sign	5.43 μs	5.41 μs	5.47 μs	5.36 μs
CPU time for Combine	7.05 μs	6.98 μs	7.23 μs	7.05 μs
CPU time for Verify	0.04 *ms*	0.28 *ms*	2.12 *ms*	2.75 *ms*

Figure 19(b) depicts the CPU time for Verify. This cost increases linearly with n in DMP because the final proof integrates n signatures from every machine, each contributing several nonces. On the other hand, DMP* eliminates the extra $O(n)$ nonces from the final proof. As such, the verification cost becomes independent of n and always equal to ~2.9ms. The overhead in DMP ranges between 5.2ms and 8.6ms, that is, it is up to about three times larger than that of DMP*. Once again, the gains of nonce chaining are more clear when n is large compared to n_a, n_b, since then the savings from reconstructing the $O(n)$ nonces become more pronounced.

The rest of the algorithms are common in both DMP* and DMP, and remain unaffected by n. Specifically, Update takes ~4μs, Combine consumes ~0.15ms, whereas the time for KeyGen is negligible. Evidently, all our algorithms are lightweight.

9.4. Sliding Window Queries

Our final set of experiments focuses on the evaluation of our sliding window techniques, namely W-DVS, W-DMP, and W-DDP. Similar to the cases of DVS*/DVS and DMP*/DMP, we used synthetic datasets. Note that, since we are the first to propose schemes for sliding window query authentication in the trusted client model, we have no competitors.

Evaluation of W-DVS. Table VII illustrates the costs involved in W-DVS when varying the sliding window size w and fixing the number of machines m to 100. The CPU times for Update and Sign are independent of w and hence remain constant and equal to ~3.5μs and ~5.4μs, respectively. Interestingly, even the cost for Combine is unaffected by w. This is because we implemented the optimization described in Section 6.1, which spends $O(w)$ space in order to eliminate the $O(w)$ factor from the processing cost at the server. For $w = 10{,}000$, this translates to cost reductions by four orders of magnitude, at the expense of about 100KB of main memory. Finally, the overhead of Verify increases linearly with w, since the latter determines the size of the result vector given to the client, which requires PRF evaluations and modular multiplications/additions for every vector element. However, even for the worst case where $w = 10{,}000$, the verification time is below 3ms. Moreover, although nonce chaining does not reduce the asymptotic cost, it halves the actual cost since the technique eliminates the need for $O(w)$ PRF evaluations for reconstructing the nonces (now only a constant number of nonces are present in the final proof). The cost of KeyGen is negligible and, thus, omitted.

Evaluation of W-DMP. We assess the performance of W-DMP assuming that, at every timestamp, it authenticates a product between an $n_a \times w$ matrix and $w \times n_b$ matrix. Table VIII displays the related costs when varying w and setting $n_a = n_b = 100$. The overheads of Update and Sign (~3.6μs and ~11.7μs, respectively) are lightweight and independent of w. Furthermore, the cost of Combine remains constant (and equal to ~0.52μs) due to the help of a similar storage technique to that of W-DVS, which renders the proof generation independent of w, consuming about 10KB. Of great interest is the cost of Verify (equal to ~18μs) that is also independent of w due to our chaining technique. Finally, the cost of KeyGen is negligible and thus omitted.

Table VIII. Performance of W-DMP when Varying w ($n_a = n_b = 100$)

Evaluated Cost	w = 100	w = 200	w = 300	w = 400	w = 500
CPU time for Update	3.61 μs	3.55 μs	3.66 μs	3.60 μs	3.61 μs
CPU time for Sign	11.65 μs	11.456 μs	11.86 μs	11.63 μs	11.69 μs
CPU time for Combine	0.95 μs	1.05 μs	1.59 μs	1.64 μs	1.48 μs
CPU time for Verify	3.20 *ms*	3.15 *ms*	3.27 *ms*	3.18 *ms*	3.22 *ms*

Table IX. Performance of W-DDP when Varying w

Evaluated Cost	w = 10	w = 100	w = 1000	w = 10000
CPU time for Update	3.56 μs	3.61 μs	3.54 μs	3.96 μs
CPU time for Sign	11.61 μs	11.61 μs	11.44 μs	12.28 μs
CPU time for Combine	0.52 μs	0.52 μs	0.52 μs	0.62 μs
CPU time for Verify	18.39 μs	18.32 μs	18.28 μs	22.01 μs

Evaluation of W-DDP. Table IX shows the costs of the algorithms in W-DDP as a function of vector size w. Similar to the cases of W-DVS and W-DMP, Sign and Verify in W-DDP entail constant operations, leading to times $\sim 3.6\mu s$ and $\sim 11.6\mu s$, respectively, regardless of the value of w. In addition, Combine exploits a similar data structure to that used in W-DVS and W-DMP (of size about 10KB), rendering the CPU time unaffected by w. It is noteworthy that, different from our basic DDP construction, W-DDP does not mandate the server to use any public information in order to remove "unnecessary" components from the final proof. This has three effects: (i) KeyGen is now the same as in W-DVS and W-DMP, which inflicts negligible cost; (ii) the signature and proof sizes are smaller (equal to 10 bytes instead of 64 bytes); and (iii) the cost of Combine is now lightweight (equal to $\sim 0.5\mu s$). Finally, the overhead of Verify is unaffected by w and equal to $\sim 20\mu s$.

9.5. Summary

Our experimental study confirms our claims that the constructions presented are lightweight and practical. The overheads of all basic protocols have very low streaming cost: the central Update operation is always measured in single-digit microsecond costs, corresponding to very high stream rates. The cost for Sign operations is comparable, except in the case of DMP that scales proportionally to the square root of the input size. The computation in Verify scales linearly with the size of the input. The server's overhead (Combine) is also small, and remains smaller than a second even in the computationally intensive case of DDP. Moreover, our DVS scheme is superior to PIRS in terms of client communication cost in the case of multiple machines, whereas DMP and DDP are the first secure, efficient, and scalable protocols for dynamic matrix multiplication and dynamic dot product, respectively.

In addition, we demonstrated the efficiency and practicality of our extensions. Specifically, we showed that DVS* and DMP* can lead to up to several orders of magnitude lower verification times than DVS and DMP, respectively. This comes with a small extra processing overhead in the Sign algorithm. Finally, our sliding window authentication schemes, namely W-DVS, W-DMP, and W-DDP, exhibit lightweight overheads and scale well with the sliding window size, taking advantage of our nonce chaining technique and data structure optimizations at the server.

10. CONCLUSIONS AND FUTURE WORK

In this article we addressed the problem of result authentication in stream outsourcing settings. While prior work has focused on simple `group-by, sum` queries in such scenarios, our protocols allow the authentication of several linear algebraic operators, such as sums or dot products over dynamic vectors and dynamic matrix multiplication, that are used in numerous applications over distributed data. Our experiments demonstrated that our protocols are extremely lightweight, especially for the owner in terms of running time, storage requirements, and bandwidth consumption. Moreover, our schemes offer strong cryptographic guarantees for their security. In our future work, we plan to extend our lightweight techniques to the challenging setting where clients may collude with the server to attack other clients. In this case, the owner only grants a public key to the clients, hiding his secret key.

ELECTRONIC APPENDIX

The electronic appendix to this article can be accessed in the ACM Digital Library.

REFERENCES

Daniel J. Abadi, Donald Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: A new model and architecture for data stream management. *VLDB J.* 12, 2, 120–139.

Shweta Agrawal and Dan Boneh. 2009. Homomorphic MACs: MAC-based integrity for network coding. In *Proceedings of the 7th International Conference on Applied Cryptography and Network Security (ACNS'09).* 292–305.

Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennfier Widom. 2003. STREAM: The stanford stream data manager (demonstration description). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03).* 665.

Mihir Bellare. 2006. New proofs for NMAC and HMAC: Security without collision-resistance. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology (CRYPTO'06).* 602–619.

Dan Boneh and David Mandell Freeman. 2011. Homomorphic signatures for polynomial functions. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT'11).* 149–168.

Paul G. Brown. 2010. Overview of SciDB: Large scale array storage, processing and analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10).* 963–968.

Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. 2009. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography (PKC'09).* 481–500.

Amit Chakrabarti, Graham Cormode, and Andrew McGregor. 2009. Annotations in data streams. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP'09).* 222–234.

Kai-Min Chung, Yael Tauman Kalai, Feng-Hao Liu, and Ran Raz. 2011. Memory delegation. In *Proceedings of the 31st Annual Conference on Advances in Cryptology (CRYPTO'11).* 151–168.

Graham Cormode, Michael Mitzenmacher, and Justin Thaler. 2012. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS'12).* 90–112.

Graham Cormode, Justin Thaler, and Ke Yi. 2011. Verifying computations with streaming interactive proofs. *Proc. VLDB Endow.* 5, 1, 25–36.

Chuck Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. 2003. Gigascope: A stream database for network applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03).* 647–651.

Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. 2003. Approximate join processing over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03).* 40–51.

Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A general purpose event monitoring system. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'07).* 412–422.

Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G. Stubblebine. 2003. Authentic data publication over the internet. *J. Comput. Secur.* 11, 3, 291–314.

Minos N. Garofalakis, Joseph M. Hellerstein, and Petros Maniatis. 2007. Proof sketches: Verifiable in-network aggregation. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'07)*. 996–1005.

Rosario Gennaro, Craig Gentry, and Bryan Parno. 2010. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the 30th Annual Conference on Advances in Cryptology (CRYPTO'10)*. 465–482.

Oded Goldreich. 2001. *The Foundations of Cryptography* - Volume 1 (Basic Techniques). Cambridge University Press.

Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. Delegating computation: Interactive proofs for muggles. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC'08)*. 113–122.

Jonathan Katz and Yehuda Lindell. 2007. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press.

Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*. 121–132.

Feifei Li, Ke Yi, Marios Hadjieleftheriou, and George Kollios. 2007. Proof-infused streams: Enabling authentication of sliding window queries on streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*. 147–158.

Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2002. TAG: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.* 36, SI, 131–146.

Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. 1996. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL.

Microsoft. 2010. StreamInsight. http://msdn.microsoft.com/en-us/library/ee362541.aspx.

Maithili Narasimha and Gene Tsudik. 2006. Authentication of outsourced databases using signature aggregation and chaining. In *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA'06)*. 420–436.

Howard Nasgaard, Bugra Gedik, Mary Komor, and Mark P. Mendell. 2009. IBM infosphere streams: Event processing for a smarter planet. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research (CASCON'09)*. 311–313.

Suman Nath and Ramarathnam Venkatesan. 2013. Publicly verifiable grouped aggregation queries on outsourced data streams. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'13)*. 517–528.

Suman Nath, Haifeng Yu, and Haowen Chan. 2009. Secure outsourced aggregation via one-way chains. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. 31–44.

HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. 2005. Verifying completeness of relational query results in data publishing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'05)*. 407–418.

HweeHwa Pang and Kian-Lee Tan. 2004. Authenticating query results in edge computing. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*. 560–571.

Stavros Papadopoulos, Graham Cormode, Antonios Deligiannakis, and Minos Garofalakis. 2013. Lightweight authentication of linear algebraic queries on data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. 881–892.

Stavros Papadopoulos, Aggelos Kiayias, and Dimitris Papadias. 2011. Secure and efficient in-network processing of exact sum queries. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'11)*. 517–528.

Stavros Papadopoulos, Yin Yang, and Dimitris Papadias. 2007. CADS: Continuous authentication on data streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*. 135–146.

Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2011. Optimal verification of operations on dynamic sets. In *Proceedings of the 31st Annual Conference on Advances in Cryptology (CRYPTO'11)*. 91–110.

Victor Shoup. 1997. Lower bounds for discrete logarithms and related problems. In *Proceedings of the 16th Annual International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'97)*. 256–266.

Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. 2003. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*. Vol. 29. 285–296.

Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. 2009. Authenticated join processing in outsourced databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*. 5–18.

Ke Yi, Feifei Li, Graham Cormode, Marios Hadjieleftheriou, George Kollios, and Divesh Srivastava. 2009. Small synopses for group-by query verification on outsourced data streams. *ACM Trans. Database Syst.* 34, 3, 15:1–15:42.

Received October 2013; revised May 2014; accepted July 2014