Lightweight Authentication of Linear Algebraic Queries on Data Streams

Stavros Papadopoulos[#] Antonios Deligiannakis[*] Graham Cormode[‡] Minos Garofalakis[*]

[#]HKUST [*]Technical University of Crete [‡]AT&T Labs-Research
stavrosp@cse.ust.hk {adeli, minos}@softnet.tuc.gr graham@research.att.com

ABSTRACT

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 easily 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* and *dot products*, as well as *dynamic matrix 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. All our schemes are very *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.

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 (DSMS), which offer efficient storage and reliable querying services to clients. Following research prototypes such as Stream [3] and Aurora [1], robust DSMS have been deployed for many applications, including IBM's InfoSphere Streams [21], Microsoft's StreamInsight [20] and AT&T's Gigascope [8]. Due to the overwhelming volume of streaming data, companies may not possess, or wish to acquire, the resources for deploying a DSMS. A practical alternative 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, e.g., 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 from 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, i.e., that they are untampered with (*integrity*) and up-to-date (*freshness*). The goal is to make stream authentication a very lightweight operation for all parties involved, and establish it as a standard tool for error-checking, in a similar way to the ubiquitous use of checksums for reliable file transfer.

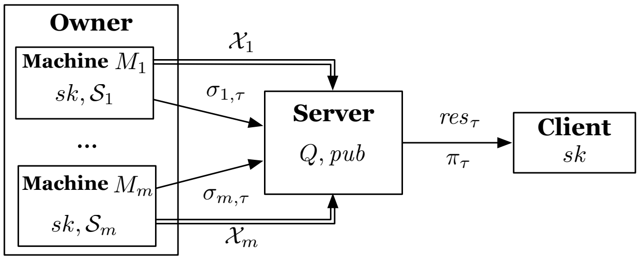

Figure 1: System setting

Targeted setting. Figure 1 illustrates our system architecture. An *owner* possesses a set of machines $M_1, M_2, \ldots, M_m$, each generating or observing a data stream $\mathcal{X}_i$ and outsourcing it to a *third party server*. These machines are not required to directly communicate with each other, and their streams are disjoint. A *client* registers a *continuous* (i.e., long-running) query Q on the union of the streams at the server. The latter periodically reports the result to the client at regular time intervals, demarcated into epochs τ.

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 that 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 above functionality for a wide range of query types, offering *cryptographic* security and satisfying certain

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

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 life-time of these systems is diminished by intense operations and communication. Secondarily, we further aim to ensure that the server's costs are also low. We next explain our contributions in detail.

Our contributions. The existing literature on stream authentication is limited in its applicability for a variety of reasons. Firstly, the range of supported queries is somewhat narrow: prior work has been primarily concerned with authenticating particular computations such as `group by,sum` queries which, while fundamental, do not cover all stream outsourcing scenarios. Secondly, the authentication cost at the owner is non-trivial; 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 for *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 extremely lightweight for the owner, as they mainly involve inexpensive hash operations and modular additions/multiplications in a very small finite field. They are also cheap 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 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 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 range 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 adapts our main schemes to a variety of important database applications in the stream authentication setting. Section 6 contains our experimental evaluation, whereas Section 7 presents concluding remarks and future directions.

2. BACKGROUND

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

Table 1: 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/vectors 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
$x \xleftarrow{\$} S$	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
$\mathsf{poly}(s)$ / $\mathsf{negl}(s)$	A positive polynomial in s / A negligible function in s
$[n]$	The set $\{1, 2, \ldots, n\}$
$F_k(x) \overset{\text{def}}{=} F(k, x)$	Pseudo-random 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

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 1 summarizes the most important notation used in this paper.

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 $\mathsf{poly}(\cdot)$ and sufficiently large s. We denote a negligible function by $\mathsf{negl}(s)$.

Pseudo-random 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 *pseudo-random function* (PRF) if for all adversaries $\mathcal{A}$ it holds

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

where $F_k(x) \overset{\text{def}}{=} F(k, x)$, $k \xleftarrow{\$} \mathcal{K}$ and $f \xleftarrow{\$} (S_1 \rightarrow S_2)$. 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 [19]. Let $\mathbb{G}$ be a group, let $p = |\mathbb{G}|$ denote the order of $\mathbb{G}$ and let $\mathbf{1}$ denote the identity element of $\mathbb{G}$. For any element $g \in \mathbb{G}$, the order of g is the least 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, the *order* of g is equal to $p = |\mathbb{G}|$ and 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 [5]. 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 $\mathcal{V} = \{g, g^k, g^{k^2}, \dots, g^{k^n}, g^{k^{n+2}}, \dots, g^{k^{2n}}\}$ where $k \xleftarrow{\$} \mathbb{Z}_p^*$, compute $g^{k^{n+1}}$. The n-DHE assumption states that, for any adversary $\mathcal{A}$, it holds:

$$\Pr[\mathcal{A}(g, g^k, g^{k^2}, \dots, g^{k^n}, g^{k^{n+2}}, \dots, g^{k^{2n}}) = g^{k^{n+1}}] \leq \mathsf{negl}(s)$$

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

One-time pad and nonces. One-time pad is a method of encrypting data which exhibits *perfect secrecy* [15], if implemented correctly. In one-time pad encryption, a message M is encrypted using a random key K which (i) has (at least) the *same* size as M, and (ii) is used exactly *once*. The encryption is performed via an 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 an XOR operation, this alternative mode of one-time pad offers the same security as the original, and it 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 [31] and SHAC [22], 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 SHAC 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 quite lightweight. On the other hand, SHAC 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, SHAC employs expensive cryptographic primitives such as modular exponentiations during authentication and verification.

Note that PIRS and SHAC 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 SHAC to our scenario, the owner *must rely on* some other message authentication technique for securely forwarding the summary to the clients via the server, e.g., using HMACs and public-key digital signatures [19], respectively. This inflicts extra overhead to 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 [2] for network coding applications and have since been widely used, such as in [4] for evaluating multivariate polynomials on *signed* data, or in [26] 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 paper) 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 [11, 25, 24, 16, 30], as well as *continuous* range queries [17, 27]. All these methods rely on *authenticated data structures* (such as Merkle trees), which 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` [12, 26, 23]. 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).

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 [6]. 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 [7]. Quite general computations can be authenticated following a streaming pass over the data, but this can require many thousands of rounds of interaction between the parties [14].

Our work differs from these prior efforts in several important respects. Firstly, we consider fundamental problems that can be adapted to solve a wider range of important queries in stream outsourcing. Secondly, our constructions impose a very low overhead to all parties. In particular, they do not entail the costly exponentiation operations involved in SHAC, 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.

3. FORMULATION

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

3.1 Stream Authentication Protocol

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

DEFINITION 1. *A stream authentication scheme is a set of five algorithms* (KeyGen, Update, Sign, $\mathsf{Combine}$, Verify) *running in time polynomial in s and described as follows:*

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

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

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

$\pi_\tau := \mathsf{Combine}(\bigcup_{i=1}^m \{\sigma_{i,\tau}\}, \bigcup_{i=1}^m \{\mathcal{X}_i(\tau)\}, pub)$**:** *A deterministic algorithm that takes as input the union of the signatures and streams at τ and public info pub. It produces proof π_τ.*

Yes|**No** $:= \mathsf{Verify}(sk, \pi_\tau, res_\tau, \tau)$**:** *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, e.g., 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 $\sigma_{i,\tau}$, which 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 $\bigcup_{i=1}^m \{\sigma_{i,\tau}\}$, $\bigcup_{i=1}^m \{\mathcal{X}_i(\tau)\}$ 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*, i.e., it verifies w.r.t. the entire history of the data streams, not since the last successful verification.

The next definition formulates *scheme correctness*.

DEFINITION 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* $\mathsf{Combine}(\bigcup_{i=1}^m \{\sigma_{i,\tau}\}, \bigcup_{i=1}^m \{\mathcal{X}_i(\tau)\}, pub)$. *Then,* $\mathsf{Verify}(sk, \pi_\tau, res_\tau, \tau)$ *returns* **Yes** *when*

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

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 subsection.

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, i.e., data privacy is *orthogonal* to our work. Nevertheless, $\mathcal{A}$ may *tamper with* the outputs at any epoch. Our security goal against $\mathcal{A}$ is *result correctness*, which 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, which is a variation of the standard *existential unforgeability under an adaptive chosen-message attack* [15]:

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 above experiment with non-negligible probability, i.e., if it holds that

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

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, which 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, i.e.,

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

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

Figure 2 presents the DVS construction, which 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 re-using the same signature at different epochs or for different machines. All operations are performed modulo p (i.e., the results are 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 $\mathbf{k} = [k_1, \dots, k_n]$. 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 that 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*. We provide formal correctness and security guarantees for DVS below.

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

THEOREM 1. DVS *is correct.*

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 above and, hence, the output is **Yes**. □

We next state the security of DVS (the proof is in Appendix A.1).

THEOREM 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_k as an HMAC [19], 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.

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

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

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

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

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

Figure 2: The DVS construction

4.2 Dynamic Matrix Product Authentication

We focus on two machines, M_a and M_b. We consider a $n_a \times n$ matrix $\mathbf{A}$ and a $n \times n_b$ 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 product*, denoted by $\mathbf{AB}$, between $\mathbf{A}$ and $\mathbf{B}$ at every epoch τ. We term such a query as a *dynamic matrix product* query. We next present a scheme, termed as 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 $\mathbf{A} = [\mathbf{a}_1\mathbf{a}_2 \dots \mathbf{a}_n]$, where $\mathbf{a}_j$ denotes the j^{th} *column* of $\mathbf{A}$. Also let $\mathbf{B} = [\mathbf{b}_1\mathbf{b}_2 \dots \mathbf{b}_n]^{\mathrm{T}}$, where $\mathbf{b}_j$ is the j^{th} *row* of $\mathbf{B}$. Then it holds:

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

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

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

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 summaries $\mathcal{S}_a[j], \mathcal{S}_b[j]$ (i.e., one for each column of $\mathbf{A}$, and one for each row of $\mathbf{B}$), and build a summary for product $\mathbf{AB}$ by summing them up.

We assume that 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. $\mathcal{S}_a$ now 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 $\mathcal{S}_b[j]$. Therefore, the machines provide additional info (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 above, 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 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. □

```
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($\{\sigma_{a,\tau}, \sigma_{b,\tau}\}, \{\mathcal{X}_a(\tau), \mathcal{X}_b(\tau)\}, pub$)
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
```

Figure 3: The DMP construction

THEOREM 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 non-zero 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 the super-quadratic 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, i.e.,

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

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 that 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 the elements but those in the diagonal).

In order to facilitate this task, the owner provides some public info 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 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 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 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. 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, it is extremely lightweight.

```
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
```

Figure 4: The DDP construction

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

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

assuming that it has access to the set of outer product values $\mathbf{c}$. However, the server does not need to explicitly generate $\mathbf{c}$. Rather, it 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. 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 the inputs $\mathcal{X}_a$ and $\mathcal{X}_b$.

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

5. APPLICATIONS

In this section, we discuss some common queries in stream outsourcing, and explain how the constructions that 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. For brevity, we omit detailed discussion of correctness and security, which follow the pattern established by the main protocols in Section 4.

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 is 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 non-zero 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, 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 captures this scenario due to its *homomorphic property*, which 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.

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 particularly challenging, with prior work focusing on approximate results [9, 29]. 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*, for queries such as `SELECT * FROM R, S WHERE R.x = S.x`. Assume without loss of generality that the join result is given by the multiset of tuples $(t_R, t_R.x, t_S)$, where t_R is a tuple from R, t_S is a tuple from S, and $t_R.x = t_S.x$ is their common value on the join attribute x. Also suppose that the domain of the join values is $[n]$. We reduce this problem to an instance of a dot-product query, making use of a *cryptographic hash function* H, e.g., SHA-1 [19]. The inputs to our schemes are streams of $(t_R.x, t_R)$ and $(t_S.x, t_S)$ pairs originating from dynamic relations R and S, respectively. We transform an update to R of $(t_R.x, t_R)$ into a tuple in $\mathcal{X}_a$ as $(t_R.x, H_R(t_R))$, applying the hash function H to t_R. Similarly, we transform updates to S of $(t_S.x, t_S)$ into a tuple in $\mathcal{X}_b$ as $(t_S.x, H_S(t_S))$. We then run the DDP protocol over updates $(j, v) = (t_R.x, H_R(t_R)) \in \mathcal{X}_a$ at M_a, and $(j, v) = (t_S.x, H_S(t_S)) \in \mathcal{X}_b$ at M_b.

The server presents the claimed output multiset of $(t_R, t_R.x, t_S)$ tuples, along with the accompanying proof π_τ from the DDP protocol. The client computes $res_\tau = \sum_{t_R.x=t_S.x} H_R(t_R) \cdot H_S(t_S)$ within the Verify routine. The protocol is correct, since the dot product between the two vectors generated by the above transformation is exactly this res_τ value. In addition to the security of DDP, here we need to show that the adversary cannot present a result that produces the same res_τ as the actual result. In order to achieve this, we must follow the so-called *random oracle model* [15, 13]. Briefly stated, this is a proof methodology that allows us to first prove the security of the scheme considering that H is a truly random function. We can do this along the lines of our proof for DDP in Appendix A.3. Then, we substitute H with a hash function like SHA-1, and claim security due to the assumption about its cryptographic properties. We omit further details due to space constraints.

In the aforementioned problem, the goal was to authenticate the tuples produced by a 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 $(t.x, 1)$, then vectors $\mathbf{a}, \mathbf{b}$ will hold the frequencies of the relations on each join value. Therefore, the equi-join size is exactly $\mathbf{a} \cdot \mathbf{b}$.

In-network Aggregation. This 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 [18]. 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 [12, 26, 23]. 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 construction applies to this scenario as well. Its KeyGen, Update, Sign and Verify routines remain the same in this case. The main changes occur to 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 extremely lightweight for all parties involved and, hence, it is ideal for the resource constrained sensor networks.

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 which 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.

Event co-occurrence. Event monitoring applications operate on massive streams in order to find patterns or correlations between certain events [10]. 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 a $n \times n$ 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 above can be generalized to matrices with arbitrary dimensions. It is apparent that our DMP construction is directly applicable for authenticating such queries.

6. EXPERIMENTS

In this section we experimentally evaluate our basic protocols, namely DVS, DMP and DDP. We compare DVS with PIRS (specifically PIRS-1) [31], 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 that 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, DMP and DDP have no competitors.

Implementation. We implemented all protocols in C on a 2.66GHz Intel Core i7 with 4GB of RAM, running MAC OS X. We used the GMP[1] and OpenSSL[2] libraries for implementing the cryptographic operations involved. We utilized HMAC with SHA-1 [19] for the F function, which produces 20-byte outputs. We employed HMAC with SHA-1 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). In DVS, DMP and PIRS, this can be as small as 10 bytes for safeguarding against guessing attacks on the keys. On the other hand, 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 $\mathbb{Z}_p$ [19], suggesting that the size of p should be twice as long as the one that protects against simple guessing.

Table 2: 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
RSA signing (1024-bit)	1.55 ms
RSA verification (1024-bit)	82.18 μs

Table 3: 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

Furthermore, we computed the generator of the group used in DDP employing the implementation techniques included in [19]. 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 $q = 2\ell p + 1$ [19]. All computations in $\mathbb{G}$ are modulo q. Table 2 includes the average cost (over 10,000 runs) of each primitive operation entailed in the implemented protocols.

Evaluation of DVS vs. PIRS. We compared DVS with PIRS using the *World Cup Dataset*[3]. 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 3 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 non-zero elements in the result vector (which 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 in the order of a few microseconds at the owner (5.3 μs for Update and 4.8 μs for Sign), and a few milliseconds at the client (48.9 μs for Verify). The summary and proof size is negligible in DVS (10 bytes). The 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 4 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, are omitted. In PIRS, there is no Combine cost, since the

[1] `http://gmplib.org/`

[2] `http://www.openssl.com/`

[3] `http://ita.ee.lbl.gov/html/contrib/WorldCup.html`

Table 4: 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

Table 5: 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

server simply forwards m summaries and HMACs to the client. This considerably increases the total proof size to 3000 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 μs). 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 5 shows the time costs of each of the operations as n varies. The Update step is similar in all cases ($\sim$ 6 μs), 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 hundred thousands of entries, this cost is in 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 6. Here, we also generate synthetic vectors of differing sizes. Observe that there is a non-trivial setup cost for this protocol, which stems from determining a generator for $\mathbb{G}$ and computing the 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 ($n = 10000$). The cost for Verify is quite low, since it requires only a constant amount of light work for checking the proof.

Summary. Our experimental study confirms our claims that the constructions presented are lightweight and practical. The overheads of all 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, which 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. Finally, DMP and DDP are the first secure, efficient, and scalable protocols for the problems of dynamic matrix multiplication and dynamic dot product, respectively.

Table 6: 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

7. CONCLUSIONS AND FUTURE WORK

In this paper we addressed the problem of result authentication in stream outsourcing settings. While prior work has focused on simple group by and 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, which are used in numerous applications over distributed data. Our experimental evaluation 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.

8. REFERENCES

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik. Aurora: A Data Stream Management System. In *SIGMOD*, 2003.

[2] S. Agrawal and D. Boneh. Homomorphic MACs: MAC-Based Integrity for Network Coding. In *ACNS*, 2009.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager (demonstration description). In *SIGMOD*, 2003.

[4] D. Boneh and D. M. Freeman. Homomorphic Signatures for Polynomial Functions. In *EUROCRYPT*, 2011.

[5] J. Camenisch, M. Kohlweiss, and C. Soriente. An Accumulator Based on Bilinear Maps and Efficient Revocation for Anonymous Credentials. In *PKC*, 2009.

[6] A. Chakrabarti, G. Cormode, and A. McGregor. Annotations in Data Streams. In *ICALP*, 2009.

[7] G. Cormode, J. Thaler, and K. Yi. Verifying computations with streaming interactive proofs. In *VLDB*, 2012.

[8] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[9] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing Over Data Streams. In *SIGMOD*, 2003.

[10] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[11] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic Data Publication over the Internet. *J. Comput. Secur.*, 11(3):291–314, 2003.

[12] M. N. Garofalakis, J. M. Hellerstein, and P. Maniatis. Proof Sketches: Verifiable In-Network Aggregation. In *ICDE*, 2007.

[13] O. Goldreich. *The Foundations of Cryptography - Volume 1, Basic Techniques*. Cambridge University Press, 2001.

[14] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating Computation: Interactive Proofs for Muggles. In *STOC*, 2008.

[15] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007.

[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic Authenticated Index Structures for Outsourced Databases. In *SIGMOD*, 2006.

[17] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios. Proof-Infused Streams: Enabling Authentication of Sliding Window Queries On Streams. In *VLDB*, 2007.

[18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[19] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1996.

[20] Microsoft. StreamInsight. `http://msdn.microsoft.com/en-us/library/ee362541.aspx`, 2010.

[21] H. Nasgaard, B. Gedik, M. Komor, and M. P. Mendell. IBM Infosphere Streams: Event Processing for a Smarter Planet. In *CASCON*, 2009.

[22] S. Nath and R. Venkatesan. Publicly Verifiable Grouped Aggregation Queries on Outsourced Data Streams. In *ICDE*, 2013.

[23] S. Nath, H. Yu, and H. Chan. Secure Outsourced Aggregation via One-way Chains. In *SIGMOD*, 2009.

[24] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying Completeness of Relational Query Results in Data Publishing. In *SIGMOD*, 2005.

[25] H. Pang and K.-L. Tan. Authenticating Query Results in Edge Computing. In *ICDE*, 2004.

[26] S. Papadopoulos, A. Kiayias, and D. Papadias. Secure and Efficient In-Network Processing of Exact SUM Queries. In *ICDE*, 2011.

[27] S. Papadopoulos, Y. Yang, and D. Papadias. CADS: Continuous Authentication on Data Streams. In *VLDB*, 2007.

[28] V. Shoup. Lower Bounds for Discrete Logarithms and Related Problems. In *EUROCRYPT*, 1997.

[29] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, 2003.

[30] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated Join Processing in Outsourced Databases. In *SIGMOD*, 2009.

[31] K. Yi, F. Li, G. Cormode, M. Hadjieleftheriou, G. Kollios, and D. Srivastava. Small Synopses for Group-by Query Verification on Outsourced Data Streams. *TODS*, 34(3), 2009.

APPENDIX

A. SECURITY PROOFS

We will prove all our constructions under the following general methodology [13]. We will construct an *ideal* version of each scheme, where function F is substituted by a truly random function f. The adversary will be able to obtain outputs of f for inputs of its choice, but it will not be given direct access to f itself. We will next prove that the scheme is secure under this ideal model. Finally, we will conclude that the *real* implementation of the scheme (i.e., the one that uses the PRF F instead of f) is secure, since otherwise the adversary would distinguish the PRF from a truly random function.

A.1 Proof of Theorem 2 (DVS)

Let $\mathbf{k} = [k_1, k_2, \ldots, k_n]$, where k_j is generated as in Update, but now via truly random function f instead of PRF F. The adversary $\mathcal{A}$ interacts with the oracle as outlined in $\mathsf{Exp}_{\mathcal{A}}$ in Section 3.2, and obtains a *single* signature for every $(i,\, \tau')$ pair for any stream of its choice, such that $\tau' \neq \tau$. Let $\mathbf{a}^*_{i,\tau'}$ be the dynamic vector produced by machine M_i at τ' for an arbitrary stream selected by the adversary. Note that $\mathcal{A}$ can query the oracle for $\mathsf{poly}(s)$ different epochs τ'. Then, $\mathcal{A}$ obtains the following set:

$$\begin{aligned}\mathcal{V} = \{\mathbf{a}^*_{i,\tau'}, (\mathbf{k}\cdot\mathbf{a}^*_{i,\tau'} + r_{i,\tau'})\}&_{(i\in[m])\wedge(\tau'\neq\tau)}\\ \cup\, \{\mathbf{a}_{i,\tau}, (\mathbf{k}\cdot\mathbf{a}_{i,\tau} + r_{i,\tau})\}&_{i\in[m]}\end{aligned}$$

where $\mathbf{a}_{i,\tau}$ is the *real* dynamic vector produced by machine M_i at epoch τ when the stream is the actual $\mathcal{X}_i(\tau)$. All the r values are constructed as in Sign, but again via function f instead of F. Set $\mathcal{V}$ is the *view* of the adversary in the attack.

$\mathcal{A}$ succeeds in the attack if it presents $(\mathit{res}^*_\tau, \pi^*_\tau)$, such that res^*_τ is *different* from the *actual* result $\mathit{res}_\tau = \sum_{i=1}^m \mathbf{a}_{i,\tau}$, and

$$\pi^*_\tau = \mathbf{k}\cdot \mathit{res}^*_\tau + \sum_{i=1}^m r_{i,\tau}$$

so that Verify outputs **Yes**. Observe that this is equivalent to finding a pair $(\mathbf{v}^*, \mathbf{k}\cdot\mathbf{v}^*)$ from $\mathcal{V}$, and then computing $\mathit{res}^*_\tau = \mathit{res}_\tau + \mathbf{v}^*$ and $\pi^*_\tau = \pi_\tau + \mathbf{k}\cdot\mathbf{v}^*$ (note that $\mathbf{v}^*$ must be *non-zero*).

Nevertheless, every $r_{i,\tau'}, r_{i,\tau}$ used in the components of $\mathcal{V}$ are *random* (due to f) and used *only once*. As such, every $(\mathbf{k}\cdot\mathbf{a}^*_{i,\tau'} + r_{i,\tau'})$ and $(\mathbf{k}\cdot\mathbf{a}_{i,\tau} + r_{i,\tau})$ in $\mathcal{V}$ can be considered as the encryption of $\mathbf{k}\cdot\mathbf{a}^*_{i,\tau'}, \mathbf{k}\cdot\mathbf{a}_{i,\tau}$ with keys $r_{i,\tau'}$ and $r_{i,\tau}$, respectively, where the encryption scheme is a straightforward variant of the *one-time pad* (see Section 2.1). Due to this scheme and since $\mathbf{k}$ is random and secret, the adversary computes $\mathbf{k}\cdot\mathbf{v}^*$ for any given $\mathbf{v}^*$ from $\mathcal{V}$ only with probability $\mathsf{negl}(s)$.

Based on the above discussion, $\mathcal{A}$ can only *guess* a pair $(\mathit{res}^*_\tau, \pi^*_\tau)$. Consider the following *multivariate polynomial* in *finite field* $\mathbb{Z}_p$

$$P(X_1, X_2, \ldots, X_{n+1}) = k_1X_1 + \ldots + k_nX_n - X_{n+1} + \sum_{i=1}^m r_{i,\tau}$$

Notice that $\mathcal{A}$ guesses $(\mathit{res}^*_\tau, \pi^*_\tau)$ correctly if and only if $\mathit{res}^*_\tau = [v^*_1, v^*_2, \ldots, v^*_n]$ such that $P(v^*_1, v^*_2, \ldots, v^*_n, \pi^*_\tau) = 0$. However, due to Lemma 1 in [28], for any (non-zero) multivariate polynomial P in $\mathbb{Z}_p$ of degree d (in our case $d = 1$) and randomly chosen $v^*_1, v^*_2, \ldots, v^*_n, \pi^*_\tau$, the probability that $P(v^*_1, v^*_2, \ldots, v^*_n, \pi^*_\tau) = 0$ is $d/p \overset{d=1}{=} 1/p = \mathsf{negl}(s)$.

We derive that DVS is *secure* w.r.t. $\mathsf{Exp}_{\mathcal{A}}$ in the *ideal model*. Therefore, we also conclude that DVS is secure in the *real model*, under the assumption that F is a PRF. $\square$

A.2 Proof of Theorem 4 (DMP)

Let $\mathbf{k}_a = [k_{a,1}, \ldots, k_{a,n_a}]$ and $\mathbf{k}_b = [k_{b,1}, \ldots, k_{b,n_b}]$, where $k_{a,i}, k_{b,i}$ are created as in Update, but now via truly random function f instead of PRF F. Also define $n_a \times n_b$ matrix $\mathbf{K} = \mathbf{k}_a \otimes \mathbf{k}_b$, such that $\mathbf{K}[i][j] = k_{a,i} \cdot k_{b,j}$. The adversary $\mathcal{A}$ interacts with the oracle as outlined in $\mathsf{Exp}_{\mathcal{A}}$ in Section 3.2, and obtains a *single* signature per every (i, τ') pair for any stream of its choice, such that $\tau' \neq \tau$. Let $\mathbf{A}^*_{\tau'}$ (respectively $\mathbf{B}^*_{\tau'}$) be the dynamic matrix produced by machine M_a (M_b) at τ' for an arbitrary stream selected by the adversary. Note that $\mathcal{A}$ can query the oracle for $\mathsf{poly}(s)$ different epochs τ'. Then, $\mathcal{A}$ obtains the set:

$$\begin{aligned}
\mathcal{V} = \{&\mathbf{a}^*_{j,\tau'}, (\mathbf{k}_a \cdot \mathbf{a}^*_{j,\tau'} + r_{a,\tau'}[j])\}_{(j\in[n])\wedge(\tau'\neq\tau)} \\
&\cup\ \{\mathbf{b}^*_{j,\tau'}, (\mathbf{k}_b \cdot \mathbf{b}^*_{j,\tau'} + r_{b,\tau'}[j])\}_{(j\in[n])\wedge(\tau'\neq\tau)} \\
&\cup\ \{\mathbf{k}_a \cdot \mathbf{a}^*_{j,\tau'} \cdot r_{b,\tau'}[j] + \rho_{a,\tau'}[j]\}_{(j\in[n])\wedge(\tau'\neq\tau)} \\
&\cup\ \{\mathbf{k}_b \cdot \mathbf{b}^*_{j,\tau'} \cdot r_{a,\tau'}[j] + \rho_{b,\tau'}[j]\}_{(j\in[n])\wedge(\tau'\neq\tau)} \\
&\cup\ \{\mathbf{a}_{j,\tau}, (\mathbf{k}_a \cdot \mathbf{a}_{j,\tau} + r_{a,\tau}[j])\}_{j\in[n]} \\
&\cup\ \{\mathbf{b}_{j,\tau}, (\mathbf{k}_b \cdot \mathbf{b}_{j,\tau} + r_{b,\tau}[j])\}_{j\in[n]} \\
&\cup\ \{(\mathbf{k}_a \cdot \mathbf{a}_{j,\tau} \cdot r_{b,\tau}[j] + \rho_{a,\tau}[j])\}_{j\in[n]} \\
&\cup\ \{\mathbf{k}_b \cdot \mathbf{b}_{j,\tau} \cdot r_{a,\tau}[j] + \rho_{b,\tau}[j]\}_{j\in[n]}
\end{aligned}$$

where $\mathbf{a}^*_{j,\tau'}$ (respectively $\mathbf{b}^*_{j,\tau'}$) is the j^{th} column (row) of $\mathbf{A}^*_{\tau'}$ ($\mathbf{B}^*_{\tau'}$) at τ', and $\mathbf{a}_{j,\tau}$ ($\mathbf{b}_{j,\tau}$) is the j^{th} column (row) of the *real* $\mathbf{A}_\tau$ ($\mathbf{B}_\tau$) produced by M_a (M_b) at τ. All the r and ρ values are constructed as in Sign, but again via function f instead of F. Set $\mathcal{V}$ is the *view* of the adversary in the attack.

Suppose that the adversary presents $(\mathit{res}^*_\tau, \pi^*_\tau)$ in the end of the attack, such that res^*_τ is *different* from the *actual* result $\mathit{res}_\tau = \mathbf{A}_\tau \mathbf{B}_\tau$. Let $\mathbf{K} : \mathit{res}^*_{\tau'} = \sum_{i\in[n_a],j\in[n_b]} \mathbf{K}[i][j] \cdot \mathit{res}^*_{\tau'}[i][j]$ denote the *Frobenius product* between $\mathbf{K}$ and $\mathit{res}^*_{\tau'}$. Then, notice that $\mathcal{A}$ succeeds in the attack if

$$\pi^*_\tau = \mathbf{K} : \mathit{res}^*_\tau + \textstyle\sum_{j=1}^{n} (r_{a,\tau}[j] \cdot r_{b,\tau}[j] - \rho_{a,\tau}[j] - \rho_{b,\tau}[j])$$

so that Verify outputs **Yes**. Observe that this is equivalent to finding a pair $(\mathbf{V}^*, \mathbf{K} : \mathbf{V}^*)$ from $\mathcal{V}$, and then computing $\mathit{res}^*_\tau = \mathit{res}_\tau + \mathbf{V}^*$ and $\pi^*_\tau = \pi_\tau + \mathbf{K} : \mathbf{V}^*$ (note that $\mathbf{V}^*$ must be *non-zero*).

We divide $\mathcal{V}$ into two subsets: $\mathcal{V}_1$ that includes the components incorporating ρ values; and $\mathcal{V}_2 = \mathcal{V} \setminus \mathcal{V}_1$. The first key observation is that every $\rho_{a,\tau'}[j]$, $\rho_{a,\tau}[j]$, $\rho_{b,\tau'}[j]$, $\rho_{b,\tau}[j]$ used in the components of $\mathcal{V}_1$ are *random* (due to f) and used *only once*. Similar to the discussion in Appendix A.1 for DVS, these serve as keys for *one-time pad* encryption and, thus, *no function* can be computed by $\mathcal{A}$ on $\mathbf{k}_a$, $\mathbf{k}_b$, $r_{a,\tau'}[j]$, $r_{a,\tau}[j]$, $r_{b,\tau'}[j]$, $r_{b,\tau}[j]$ from $\mathcal{V}_1$ with non-negligible probability.

The second observation is that, based on the above discussion, values $r_{a,\tau'}[j]$, $r_{a,\tau}[j]$, $r_{b,\tau'}[j]$, $r_{b,\tau'}[j]$ appear *random* in view $\mathcal{V}_1$ of $\mathcal{A}$. Moreover, observe that they are used only *once* in the components of $\mathcal{V}_2$. Hence, they can also be regarded as *one-time pad* keys for the components in $\mathcal{V}_2$. This means that *no function* can be computed by $\mathcal{A}$ on $\mathbf{k}_a$, $\mathbf{k}_b$ (and, hence, also on $\mathbf{K} = \mathbf{k}_a \otimes \mathbf{k}_b$) from $\mathcal{V}_2$ with non-negligible probability. We conclude that $\mathcal{A}$ computes $\mathbf{K} : \mathbf{V}^*$ for any given $\mathbf{V}^*$ from $\mathcal{V}$ with probability $\mathsf{negl}(s)$.

Similar to the case of DVS, $\mathcal{A}$ can only *guess* pair $(\mathit{res}^*_\tau, \pi^*_\tau)$. Consider the following *multivariate polynomial* in *finite field* $\mathbb{Z}_p$

$$\begin{aligned}
P(X_1, \ldots, X_{n_a\cdot n_b+1}) = {}& k_{a,1} \cdot k_{b,1} \cdot X_1 + \ldots \\
&+ k_{a,n_a} \cdot k_{b,n_b} \cdot X_{n_a\cdot n_b} - X_{n_a\cdot n_b+1} \\
&+ \sum_{j=1}^{n} (r_{a,\tau}[j] \cdot r_{b,\tau}[j] - \rho_{a,\tau}[j] - \rho_{b,\tau}[j])
\end{aligned}$$

Notice that $\mathcal{A}$ guesses $(\mathit{res}^*_\tau, \pi^*_\tau)$ correctly if and only if $\mathit{res}^*_\tau = [v^*_1, v^*_2, \ldots, v^*_{n_a\cdot n_b}]$ such that $P(v^*_1, v^*_2, \ldots, v^*_{n_a\cdot n_b}, \pi^*_\tau) = 0$. Nevertheless, due to Lemma 1 in [28], for any (non-zero) multivariate polynomial P in $\mathbb{Z}_p$ of degree d (again, in our case $d = 1$) and randomly chosen v^*_1, v^*_2, …, $v^*_{n_a\cdot n_b}$, π^*_τ, the probability that $P(v^*_1, v^*_2, \ldots, v^*_{n_a\cdot n_b}, \pi^*_\tau) = 0$ is $d/p \overset{d=1}{=} 1/p = \mathsf{negl}(s)$.

We derive that DMP is *secure* w.r.t. $\mathsf{Exp}_{\mathcal{A}}$ in the *ideal model*. Therefore, we also conclude that DMP is secure in the *real model*, under the assumption that F is a PRF. □

A.3 Proof of Theorem 6 (DDP)

Let $\mathbf{k}_a = [k, k^2, \ldots, k^n]$ and $\mathbf{k}_b = [k^n, k^{n-1}, \ldots, k]$, where k is a random value in $\mathbb{Z}^*_p$. The adversary $\mathcal{A}$ interacts with the oracle as outlined in $\mathsf{Exp}_{\mathcal{A}}$ in Section 3.2, and obtains a *single* signature per every (i, τ') pair for any stream of its choice, such that $\tau' \neq \tau$. Let $\mathbf{a}^*_{\tau'}$ (respectively $\mathbf{b}^*_{\tau'}$) be the dynamic vector produced by machine M_a (M_b) at τ' for an arbitrary stream selected by the adversary. Note that $\mathcal{A}$ can query the oracle for $\mathsf{poly}(s)$ different epochs τ'. Taking into account also pub generated in KeyGen, $\mathcal{A}$ obtains the set:

$$\begin{aligned}
\mathcal{V} = \{&\mathbf{a}^*_{\tau'}, (\mathbf{k}_a \cdot \mathbf{a}^*_{\tau'} + r_{a,\tau'}), (\mathbf{k}_a \cdot \mathbf{a}^*_{\tau'} \cdot r_{b,\tau'} + \rho_{a,\tau'})\}_{\tau'\neq\tau} \\
&\cup\ \{\mathbf{b}^*_{\tau'}, (\mathbf{k}_b \cdot \mathbf{b}^*_{\tau'} + r_{b,\tau'}), (\mathbf{k}_b \cdot \mathbf{b}^*_{\tau'} \cdot r_{a,\tau'} + \rho_{b,\tau'})\}_{\tau'\neq\tau} \\
&\cup\ \{(\mathbf{a}_\tau, \mathbf{k}_a \cdot \mathbf{a}_\tau + r_{a,\tau}), (\mathbf{k}_a \cdot \mathbf{a}_\tau \cdot r_{b,\tau} + \rho_{a,\tau})\} \\
&\cup\ \{\mathbf{b}_\tau, (\mathbf{k}_b \cdot \mathbf{b}_\tau + r_{b,\tau}), (\mathbf{k}_b \cdot \mathbf{b}_\tau \cdot r_{a,\tau} + \rho_{b,\tau})\} \\
&\cup\ \{g^{k^j}\}_{j\in[2n]\setminus\{n+1\}}
\end{aligned}$$

where $\mathbf{a}_\tau$ (respectively $\mathbf{b}_\tau$) is the *real* dynamic vector produced by machine M_a (M_b) at epoch τ. All the r and ρ values are constructed as in Sign, but again via function f instead of F. The set $\mathcal{V}$ is the *view* of the adversary in the attack.

Suppose that the adversary presents $(\mathit{res}^*_\tau, \pi^*_\tau)$ in the end of the attack, such that res^*_τ is *different* from the *actual* result $\mathit{res}_\tau = \mathbf{a}_\tau \cdot \mathbf{b}_\tau$. Then, notice that $\mathcal{A}$ succeeds in the attack if

$$\pi^*_\tau = g^{k^{n+1} \cdot \mathit{res}^*_{\tau'} + (r_{a,\tau} \cdot r_{b,\tau} - \rho_{a,\tau} - \rho_{b,\tau})}$$

so that Verify outputs **Yes**. Observe that this is equivalent to finding a pair $(v^*, k^{n+1} \cdot v^*)$ from $\mathcal{V}$, and then computing $\mathit{res}^*_\tau = \mathit{res}_\tau + v^*$ and $\pi^*_\tau = \pi_\tau \cdot g^{k^{n+1} \cdot v^*}$ (note that v^* must be *non-zero*).

We divide $\mathcal{V}$ into two subsets: $\mathcal{V}_1$ that includes $\{g^{k^j}\}_{j\in[2n]\setminus\{n+1\}}$, and subset $\mathcal{V}_2 = \mathcal{V} \setminus \mathcal{V}_1$. Following a similar argumentation as in the case of DMP in Appendix A.2, due to its equivalence to a *one-time pad*, the adversary cannot extract any information about $\mathbf{k}_a$ and $\mathbf{k}_b$ (and, hence, also for k^{n+1}) from $\mathcal{V}_2$ with non-negligible probability. Moreover, due to the n-DHE assumption (Section 2.1), $\mathcal{A}$ can compute $g^{k^{n+1}}$ from $\mathcal{V}_1$ only with $\mathsf{negl}(s)$ probability. We conclude that $\mathcal{A}$ finds a pair $(v^*, k^{n+1} \cdot v^*)$ from the entire $\mathcal{V}$ with probability $\mathsf{negl}(s)$.

Thus, similar to the case of DVS and DMP, $\mathcal{A}$ can only *guess* a pair $(\mathit{res}^*_\tau, \pi^*_\tau)$. Consider the following *multivariate polynomial* in the *finite field* $\mathbb{Z}_p$

$$P(X_1, X_2) = k^{n+1} \cdot X_1 - X_2 + (r_{a,\tau} \cdot r_{b,\tau} - \rho_{a,\tau} - \rho_{b,\tau})$$

Let $\pi^*_\tau = g^{x^*_\tau}$. For random π^*_τ, x^*_τ is also random. $\mathcal{A}$ guesses $(\mathit{res}^*_\tau, \pi^*_\tau)$ correctly if and only if $P(\mathit{res}^*_\tau, x^*_\tau) = 0$. Due to Lemma 1 in [28], for any (non-zero) multivariate polynomial P in $\mathbb{Z}_p$ of degree d (in our case $d = 1$) and randomly chosen res^*_τ, x^*_τ, the probability that $P(\mathit{res}^*_\tau, x^*_\tau) = 0$ is $d/p \overset{d=1}{=} 1/p = \mathsf{negl}(s)$.

We derive that DDP is *secure* w.r.t. $\mathsf{Exp}_{\mathcal{A}}$ in the *ideal model*, under the n-DHE assumption. Therefore, we also conclude that DDP is secure in the *real model*, under the n-DHE assumption and the assumption that F is a PRF. □