Programming Support for an Integrated Multi-Party Computation and MapReduce Infrastructure

Nikolaj Volgushev, Andrei Lapets, Azer Bestavros
Email: {nikolaj, lapets, best}@bu.edu
CS Dept., Boston University
111 Cummington Mall
Boston, MA USA 02215

***Abstract*—We describe and present a prototype of a distributed computational infrastructure and associated high-level programming language that allow multiple parties to leverage their own computational resources capable of supporting MapReduce [1] operations in combination with multi-party computation (MPC). Our architecture allows a programmer to author and compile a protocol using a uniform collection of standard constructs, even when that protocol involves computations that take place locally within each participant's MapReduce cluster as well as across all the participants using an MPC protocol. The high-level programming language provided to the user is accompanied by static analysis algorithms that allow the programmer to reason about the efficiency of the protocol before compiling and running it. We present two example applications demonstrating how such an infrastructure can be employed.**

I. Introduction

Cloud computing makes it possible to collocate the virtualized resources and data assets that belong to multiple institutions, organizations, or agencies. Such collocation facilitates the development of novel applications and services that leverage very large data sets from a variety of sources. Examples are paramount ranging from smart-city [2], to genomics [3], to cybersecurity [4] applications.

A major hurdle in truly unleashing the potential of big-data analytics is the justified concern related to proprietary data sets. Thus, while feasible in a collocated setting, the development of a number of big-data applications of tremendous benefit to individual organizations and/or to society at large remain elusive. Two specific examples that we target in related projects are the use of payroll data from multiple institutions to shed light on pay inequities [5], [6] and the use of corporate network data to identify global advanced persistent threats [7].

To facilitate the development of applications that leverage private and possibly highly-sensitive data assets from multiple organizations or agencies, there is a need to extend popular, scalable data analytics platforms to allow for computation to be done without requiring constituent organizations or agencies to *share* or *release* their data assets. Towards that goal, secure Multi-Party Computation (MPC) is a promising approach that allows a group of organizations (or parties) to jointly perform a desirable computation without having to release any of the privately held data assets on which the computation is to be performed. More specifically, secure MPC allows a set of parties to compute a function over data they individually hold (the private inputs) while ensuring that the only information that can be gleaned is the result of the function evaluation as opposed to the private inputs to this function (unless such function *leaks* these inputs).

Secure MPC has been an active area of cryptography research for over 30 years [8]. While incredibly powerful (and elegant), secure MPC has remained mostly a subject of academic interest, with significant advances made in the last few years to bridge the theoretical underpinnings of MPC to its applied aspects (e.g., efficient computation of specific functions or algorithms) [9]–[13].

We approach this problem from a different perspective– that of practical software development in a cloud setting. Rather than focusing on efficient implementations of MPC primitives or functionalities in isolation from prevalent software development platforms, we focus on extending one of the most popular cloud software development platforms with constructs that integrate MPC into the platform's programming paradigm. More specifically, we extend implementations of MapReduce,[1] arguably the most popular cloud-based analytics programming paradigm, with constructs that allow programmers to leverage MPC for analytics that require access to data assets (and in general virtualized cloud resources) that are spread out across (and under the strict control of) multiple organizations.

Towards that end, in this paper, we propose a distributed computational infrastructure and associated high-level programming language that allow multiple parties to leverage their own computational resources capable of supporting MapReduce [1] operations in combination with multi-party computation (MPC).

Our approach to integrating the MPC and the MapReduce platforms recognizes (and supports) an important fact: Decisions related to constraints on data sharing across institutional boundaries are orthogonal to the analytics that needs to be done on that data. Indeed, these constraints may not even be known to the developers of these analytics. Thus, the architecture we present in this paper allows programmers to add MPC constructs to an existing

[1] The MapReduce programming paradigm [1] facilitates processing of large data sets using elastic (virtualized) clusters managed using Hadoop [14] and Apache Spark [15] systems.

MapReduce code base, which was developed without explicit specification of what part of the overall computation will need to take place locally within each participant's MapReduce cluster, or across all the participants using an MPC protocol.

Another differentiating characteristic of our approach is the creation of a single, uniform language based on MapReduce constructs that compiles into MPC and/or MapReduce operations. Having such a language allows us to use static analysis to infer (and expose to programmers) important characteristics related to the efficiency and performance of resulting MPC protocols. This provides programmers with the ability to compare and contrast different design decisions – e.g., the implications from doing various Map and Reduce operations in various orders, or of using different key-value pairs, etc.[2]

The remainder of the paper is organized as follows. Section II summarizes related work on MPC platforms. Section III describes the architecture and implementation of our integrated backend component. Section IV defines the high-level programming language and associated static analysis and compilation methods. Section V provides two examples illustrating how the overall system can be used, and Section VI concludes with a summary of our ongoing and future work.

II. Related Work

In this section, we present a brief summary of recent efforts aiming at making multi-party computation practical and accessible to programmers.[3] Current public MPC implementations include Viff [10], Sharemind [11], ObliVM [12], and GraphSC [13]. In contrast to the work we present in this paper, these frameworks are concerned with implementing MPC in isolation, i.e., without integration into a scalable platform that admits non-MPC operations.

Viff [10] implements an asynchronous protocol for general multi-party computation over arithmetic circuits. The protocol is based on Shamir and pseudo random secret sharing and provides runtimes for dishonest minority as well as active adversaries. The framework is implemented in Python using the Twisted web framework for asynchronous participant communication. Sharemind [11] implements a custom secret sharing scheme and share computing protocols over arithmetic circuits. Sharemind is restricted to three participants, a passive adversary and requires an honest majority. Sharemind provides a custom, C-like programming language for implementing MPC tasks and has been deployed to implement production-level applications for financial data analysis [16]. ObliVM [12] supports secure two party computation and a domain-specific language for compiling high-level programming abstractions such as MapReduce operations to oblivious representations of these abstractions. ObliVM implements a garbled circuit backend but allows for the addition of other MPC protocols. GraphSC [13] extends ObliVM with an API for graph-based algorithms and adds parallelization to the evaluation of the resulting oblivious algorithms.

III. Integrating MPC and MapReduce

Our infrastructure targets a scenario in which multiple parties, each having their own computational infrastructure capable of performing MapReduce operations, wish to execute a single protocol that may involve both local computations using MapReduce and computations across all the parties using an MPC protocol. While our prototype implementation utilizes specific implementations of MapReduce and MPC, there is no reason that different implementations of each cannot be used in such an infrastructure (and, in fact, it may be desirable to use multiple implementations, as each may have its own unique properties or performance characteristics).

The participating parties (or an external third party) can implement the desired protocol in our high-level programming language. The protocol is compiled into a job specification consisting of MapReduce tasks to be executed by each party locally, MPC tasks to be executed across parties, and a schedule determining the task execution order. Our prototype implementation provides an execution environment for such job specifications.

We describe our prototype implementation by giving a brief overview of the design and providing concrete implementation details. We also describe an example deployment scenario for our prototype and detail how a program implemented in our high-level language is compiled and executed on the deployed platform.

A. Overview

Our prototype platform is comprised of two types of components: *controller nodes* and *worker nodes*. Each party participating in a protocol execution controls a worker node. A worker node accesses a party's private data, acts as a driver for local MapReduce tasks, and participates in MPC tasks across parties. A controller node oversees the execution of the job. It distributes the appropriate tasks to each worker node, enforces a synchronized execution of these tasks (which is critical for MPC tasks that require the simultaneous collaboration of multiple worker nodes), and provides a storage medium that can be used to share public data across worker nodes.

B. Implementation Details

The controller and worker node software is implemented using Python (this software is available as an AMI on AWS). The controller node network interface is a web server implemented in the Python Twisted Web framework and serves content over HTTP (we plan to update the controller to serve HTTPS content in the future).

Each worker node is equipped with a MapReduce component, an MPC component, an interface to access private storage, and a network interface for communicating with the controller node. The network interface is implemented as a simple web client. The MapReduce component

[2] While we do not address it in this paper, assisting programmers with questions related to security and/or correctness of MPC protocols is a separate and important concern that the use of a uniform language makes possible.

[3] A comprehensive review of MPC literature and of cloud platforms that enable scalable analytics is beyond the scope of this paper.

provides an interface for connecting to an Apache Spark cluster and submitting tasks specified in PySpark (Spark's Python API). The MPC component configures a worker node to participate in the Viff [10] MPC protocol with the other worker nodes, and provides an interface for running MPC tasks specified in Viff. The MapReduce and MPC components act as integration points between our platform and specific MapReduce and MPC implementations. We have defined a general API for both components with the goal of extending support to other MapReduce and MPC backends in the future.

All communication between controller and worker nodes is implemented as JSON messages over HTTP (future versions of the prototype will communicate over HTTPS). The job specification and all other configuration objects are JSON encodings of Python class instances.

We now concretize the above details by giving an example deployment scenario of our platform.

C. Example Deployment

Figure 1 (left) illustrates an example prototype deployment. In this scenario, three parties with existing Apache Spark infrastructures and available EC2 resources wish to run a computation consisting of local MapReduce operations and operations ranging across parties while preserving the privacy of their data. We provide two concrete examples of such applications in Section V. The three parties designate a service broker as an impartial party to administer the overall execution of the computation.

The service broker launches a controller node with access to an S3 storage instance to serve as shared storage to all parties. Each participating party launches a worker node and configures it with access to a private S3 storage instance, the master node of an Apache Spark cluster, and the network address of the controller node.[4]

Upon completing the above preconfiguration steps, each worker node automatically registers with the controller node and obtains a global MPC configuration specifying the network and security parameters of all other participating worker nodes.

D. Program Compilation

Programs specified in the high-level programming language described in Section IV can be compiled and executed on a configured platform. Given a high-level program and a specification of the target backend frameworks, the compiler produces a job specification consisting of a collection of tasks and a task schedule. The tasks consist of an encoding of the required MapReduce operations as well as input and output handlers. The encoding is specific to the target backend framework.

Figure 1 (right) illustrates an example compilation. In this scenario the target frameworks of the compilation are Apache Spark and Viff. The program compiles to two tasks that are to execute consecutively, starting with the MapReduce task. The MapReduce task is encoded as a Python module implementing the required MapReduce operations, input operations, and output operations in Spark's PySpark API. Similarly the MPC task is encoded as a Python module implementing the required operations in Viff.

A job specification produced by the compiler can be executed on a configured platform such as the one shown in III-C. We give a concrete example of a high-level program and the resulting task-level encodings in Section V.

E. Job Execution

Each worker node obtains the job specification from the controller. Note that all worker nodes receive the same job specification (i.e., all worker nodes receive the same tasks and task schedule).

Worker nodes execute tasks in the order specified by the task schedule. The task schedule can include branching decisions driven by the results of previous tasks. All branching decisions are evaluated by worker nodes locally. In our current implementation, we restrict branching to be based only on data that is available and identical across all worker nodes (e.g., the result of an MPC task). Since each MPC task requires the participation of all worker nodes, diverging control flow could lead to a deadlock of the system if different worker nodes attempt to execute different MPC tasks.

Before evaluating the next local MapReduce or MPC task, each worker node registers with the controller node and defers task execution until all other worker nodes have registered. This synchronization barrier ensures the availability of all worker nodes for MPC tasks.

The evaluation of a local MapReduce task consists of reading data from the private and/or shared key-value store, performing the specified MapReduce operations on it and updating the key-value stores with the result. These operations are specified in the native language of the MapReduce backend (i.e., in Python, using the PySpark API in the case of Apache Spark).

The evaluation of an MPC task follows the same general structure of reading, processing, and writing key-value store data. Before data can be processed, however, it must be distributed across all parties in a privacy-preserving format. Viff (as well as many other MPC frameworks) achieves this via secret sharing. As part of integrating Viff with our platform we implemented methods for distributing key-value stores by broadcasting the keys (we assume keys to be public) and secret sharing the corresponding values across the participating parties. The data processing steps are expressed as Map and Reduce operations. This requires support for such operations from the MPC backend. In the case of Viff, which is implemented in Python and uses operator overloading to implement arithmetic operations over secret shared values, we were able to use Python's built-in functions directly.

At the end of an MPC task execution, the results that are in the form of secret shares must be opened and

[4] Note that these preconfiguration steps can be fully automated given the participating parties' AWS credentials, the network addresses of each Spark cluster, the S3 storage instances, and the controller node.

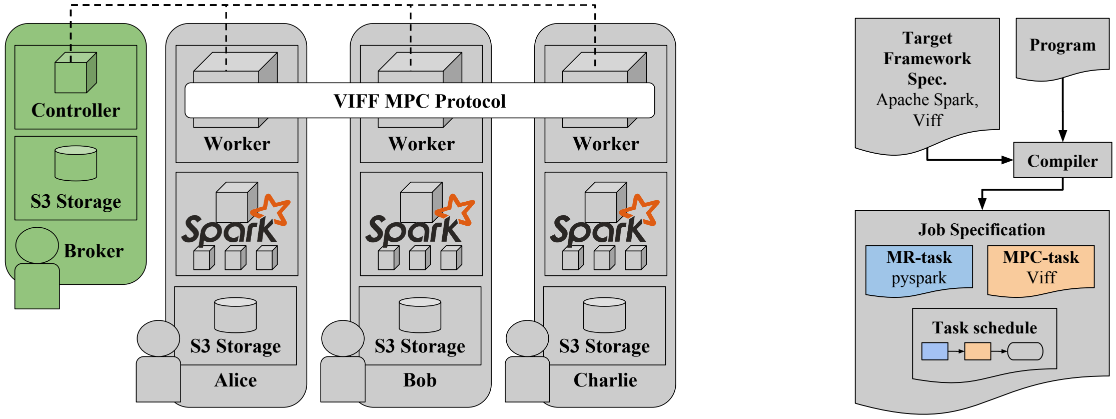

Fig. 1. Prototype deployment scenario in a service brokerage setting (left) and compilation workflow of a high-Level program (right).

revealed to the appropriate parties. This can be specified within a high-level protocol definition using the **gather** construct, defined in Section IV. Each party broadcasts the keys of the secret shared result values it requires. All parties holding a share of the value must participate for the share to be successfully reassembled. This prevents a single party from acquiring results without all other parties' consent.

Once all tasks in the task schedule have finished, the final result is reported to the target specified in the high-level protocol definition. This could be the controller node, the worker nodes, or a specific subset of worker nodes, depending on the specific application.

IV. High-Level Language

Table I lists the syntax of the high-level programming language. A protocol defined using the language consists of a sequence of statements (each of which is either a variable assignment or one of a few very basic constructs for branching, looping, and named procedures). The statements could be viewed as stages within the overall protocol. Expressions represent data (whether local to a participant or globally available to all participants) and the computations that are performed at each stage of the protocol on that data. These expressions are built up using operators that closely resemble those supported by a MapReduce infrastructure, as well as a few new operators that bridge the gap between a MapReduce computation and an MPC computation. For a more thorough presentation of the semantics of the language and the static analysis algorithms defined over it, we refer the reader to a more extensive technical report [17].

A. Language Semantics

Semantically, every expression in the language can be considered a key-value store: a collection of tuples of the form (k, v) where k must be a key (i.e., a value of a primitive type such as an integer or string) and v can be any value (including, potentially, another nested tuple). Integer, string, and boolean constants can thus be viewed as key-value stores of the form $[((),\ c)]$. The logical, relational, and arithmetic operators such as $+$ and $==$ are defined on singleton key-value stores of this form, and the key is ignored (i.e., $[(\text{“}\mathbf{a}\text{”}, 1)] + [(\text{“}\mathbf{b}\text{”}, 2)]$ evaluates to $[((),\ 3)]$). It is assumed that every participant running the protocol has a local key-value store corresponding to each declaration of the form **own** x **:= store**(τ, τ) (naturally, different participants may have different keys and values within each of these). Every assignment statement specifies whether the data that results from evaluating the assigned expression is in storage local to the participant (i.e., **own**) or in storage available to everyone (i.e., **all**). Static analysis algorithms can notify the programmer if these are not used consistently; how data available to all is actually distributed and where it is stored is up to the specific compiler and backend implementation.

Two new constructs are of particular interest: **scatter** and **gather**. For any expression δ that corresponds to data available only to an individual participant, evaluating **scatter**(δ) involves taking the result of evaluating δ and turning it into an MPC share. Any subsequent operations applied to this data should be interpreted as operations on shares within the underlying MPC infrastructure. Assuming that δ represents data that is currently being shared, evaluating **gather**(δ) involves reassembling the shares and returning a result to every participant that evaluates this expression.

Note that because in our infrastructure keys are never private, it is possible for individuals to perform **filter** operations using sets of keys from their own local key-value stores (which they can obtain using **keys**$(\ldots)$).

B. Static Analysis

One benefit of having a single high-level language for the integrated infrastructure is that it is possible to define static analysis algorithms over protocols that can furnish the programmer with valuable information about the properties of the protocol before it is ever deployed and executed. This is particularly valuable in a distributed setting, as simulation or dynamic testing may be expensive or infeasible.

user type	α	$\in$	Names
type	τ	::=	**int** | **bool** | **str** | α
integer	n	$\in$	$\mathbb{Z}$
string	σ	$\in$	Strings
constant	c	::=	**true** | **false** | n | σ
variable	x	$\in$	Names
parameters	$\overline{x}$	::=	x , . . . , x
infix op.	o	::=	+ | − | * | **or** | **and**
		|	== | != | < | <=
term	t	::=	x | c | (t) | (t , t)
		|	$t\ o\ t$ | **min(** t , t **)**
keys	k	::=	() | n | σ | **keys(** δ **)**
data	δ	::=	x
		|	**scatter(** δ **)**
		|	**gather(** δ **)**
		|	**join(** δ , δ **)**
		|	**union(** δ , δ **)**
		|	**filter(** k , δ **)**
		|	**update(** δ , δ **)**
		|	**map(lambda** $\overline{x}$**:** t , δ **)**
		|	**reduce(lambda** $\overline{x}$**:** t , δ **)**
expression	e	::=	t | δ | **store(** τ , τ **)**
statement	s	::=	**type** $\alpha = \tau$
		|	**own** x **:=** e
		|	**all** x **:=** e
		|	**if** t**:** p **else:** p
		|	**repeat** n**:** p
		|	**proc** x**:** p
		|	**call** x
		|	**return** e
program	p	::=	$s_1 \ldots s_n$

TABLE I. Abstract syntax for the high-level programming language.

One consequence of employing MPC is that there is overhead both in terms of the number of operations that each participant must perform, and in terms of the quantity of messages that must be passed between participants when the protocol is executed. It is possible that a programmer who wishes to play MPC may have more flexibility with regard to privacy than with performance, and would find it useful to compare the costs associated with a protocol that employs MPC with one that does not. Using static analysis, it is possible to derive symbolic representations of cost, as in similar work in other domains [18].

For example, consider the efficiency of operations if Sharemind is used as the MPC backend component [19]. Suppose that the high-level protocol description specifies that an addition operation must be performed on secret shared integer data. In this case, it is known that this only requires every participant to perform an addition operation on their own share of that data. On the other hand, a multiplication operation may require a collection of additions and multiplications to be performed by each participant, as well as the passing of multiple messages between all the participants. By specifying a set of inference rules over expression syntax trees for recursively assembling a symbolic function in terms of the number of participants running the protocol, it is possible to define an algorithm for deriving a running time for a given expression. Such an approach is described in more detail in the extended report [17].

V. Example Applications

We present two example applications that demonstrate how our integrated infrastructure can be used to implement a protocol that employs both MapReduce and MPC capabilities.

A. Computing Aggregate Compensation Statistics

We consider a scenario in which several firms wish to compute compensation statistics broken down by category across all firms without revealing these statistics for the individual firms. This application exemplifies a real-world use of multi-party computation [6], [20].

Figure 2 shows the implementation of the protocol using the syntax defined in Table I. Figures 3 and 4 show partial compilation results for selected sections of the implementation. The compilation of the high-level protocol definition produces four tasks. A MapReduce task implements Lines 3-7, followed by an MPC task implementing Line 9. The branch is evaluated by all worker nodes (note that the value d will be the same across all workers as required by our prototype). Two MPC tasks implement the two different branches on Lines 11 and 13 respectively. Figure 3 demonstrates a portion of the first MapReduce task. A segment of the first MPC task is shown in Figure 4. For brevity, we omit the definitions of the methods `construct_entire_kv_store`, `establish_ownership`, and `open_shares_to_owners`, which rely on the existing Viff framework to facilitate secret-sharing values, broadcasting the corresponding keys across all participants, and gathering and opening shares.

B. Computing Hop Counts to Compromised Nodes

Inspired by algorithms for threat propagation [21], we consider a scenario in which members of a group of interconnected participants, each with their own private network, are interested in computing the shortest distance from each of their local network nodes to a compromised node within the overall network. Naturally, they are interested in doing so without revealing information about their own local network.

Each party holds private data describing its local network topology and which nodes within the network are compromised. The public network, the boundary nodes

```
 1: type gender = str
 2: type salary = int
 3: own data := store(gender, salary)
 4:
 5: own m := reduce(lambda x,y: x+y, filter("m", data))
 6: own f := reduce(lambda x,y: x+y, filter("f", data))
 7: own d := m - f
 8:
 9: own s := gather(reduce(lambda x,y: x+y, scatter(d)))
10: if s > 0:
11:  return gather(reduce(lambda x,y: x+y, scatter(m)))
12: else:
13:  return gather(reduce(lambda x,y: x+y, scatter(f)))
```

Fig. 2. Private computation of aggregate salary by gender.

```
data = sc.textFile(str(in_handle))\
    .map(jsonpickle.decode)
m = data.filter(lambda x: x[0] == ’m’)\
        .reduceByKey(lambda x, y: x + y)\
        .collect()
f = data.filter(lambda x: x[0] == ’f’)\
        .reduceByKey(lambda x, y: x + y)\
        .collect()
d = (’d’, m[0][1] - f[0][1])
out_handle.write(d)
```

Fig. 3. Spark encoding of Lines 3-7 in Figure 2.

```
entire_kv_store = construct_entire_kv_store(
    viff_runtime, private_kv_store, participants)
d = filter(
  lambda x: x[0] == ’d’, entire_kv_store)
sum = (’sum’, reduce(lambda x, y: x[1] + y[1], d))
owners = establish_ownership(viff_runtime,
    sum, participants)
res = open_shares_to_owners(viff_runtime,
    total_diff, participants, owners)
out_handle.write(res)
```

Fig. 4. Viff encoding of Line 9 in Figure 2.

(i.e., private nodes with edges leaving a party’s private network), and the compromised public nodes are known to all participants. Each party computes the hop counts for its private nodes locally, then employs MPC to compute the hop counts of its boundary nodes across the entire graph without revealing its hop counts to the other parties. Finally, using the boundary node hop counts, each party updates its private node hop counts locally.

VI. Conclusion

In this paper we presented what we believe to be the first integration of MPC capabilities into the widely-popular MapReduce programming paradigm. This integration allows programmers to employ the same set of constructs to specify computation that involves both MapReduce and MPC operations. More importantly, it provides programmers with a seamless way to compare and contrast the efficiency of different strategies for structuring their implementation of analytics on private data sets spread out across various administrative domains. Our on-going work is proceeding along a number of different directions.

```
type node = int
type dist = int

all pub_graph := store(node, node)
all pub_dst := store(node, dist)
all bdr_graph := store(node, node)
all graph := union(pub_graph, bdr_graph)

own cmpny_graph := store(node, node)
own cmpny_dst := store(node, dst)
own bdr_dst := store(node, dst)
own dst := union(cmpny_dst, bdr_dst)

proc local:
  repeat 100:
    own nbr_dst :=
      map(
        lambda k,v,u: (v, u+1),
        join(map(lambda x,y: (y,x), cmpny_graph), dst))
    own dst :=
      reduce(lambda x,y: min(x,y), union(dst, nbr_dst))

  own cmpny_dst := update(cmpny_dst, dst)
  own bdr_dst := update(bdr_dst, dst)

call local
all mpc_dst := scatter(bdr_dist)

repeat 100:
  all nbr_dst :=
    map(
      lambda k,v,u: (v, u+1),
      join(map(lambda x,y: (y,x), graph), mpc_dst))
  all mpc_dst :=
    reduce(
      lambda x,y: min(x,y), union(mpc_dst, nbr_dst))

own bdr_dst := gather(filter(keys(bdr_dst), mpc_dst))
own dst := union(cmpny_dst, bdr_dst)
call local
```

Fig. 5. Private computation of hop-distance from/to compromised nodes in a multi-institutional network.

First, we view the language and associated architecture presented in this paper as providing a generic framework that could support a host of different MapReduce and MPC implementations. Supporting a diversity of analytics and infrastructures is necessary since the cloud environments of multiple institutions are likely to be heterogeneous (e.g., using Hadoop versus Spark, or using different flavors of MapReduce). More importantly, by supporting multiple MPC implementations (e.g., Viff and Sharemind versus two-party computation frameworks such as ObliVM), it is possible to exploit different performance optimization opportunities, depending on the specifics of the problem at hand. Therefore, a major thrust of our work is to extend our coverage of MapReduce and MPC implementations along these lines.

Second, we are generalizing our architecture by refining the execution model of MPC tasks. Our current prototype requires that all parties actively participate in an MPC task. However, it would be beneficial to allow for configurations in which only a subset of worker nodes actively perform an MPC task, while the other worker nodes contribute their data in secret shared form. This extension will bring performance gains since the communication cost of many MPC protocols increases with the number of active participants. It will also make our platform more flexible and diverse (as discussed above), thus allowing for the support of frameworks such as Sharemind which require a fixed number of active MPC participants.

Upon implementing the above extensions, we will evaluate the performance implications of employing different backend MPC protocols and altering the number of active worker nodes, when applicable. We defer performance evaluation to this stage since measuring the performance of our current prototype would only reflect the performance guarantees of Spark and Viff, which have already been studied and presented [10], [15]. We also note that, to the best of our knowledge, there currently exist no frameworks that can be directly compared to our platform with regard to performance because existing frameworks focus either exclusively on a MapReduce implementation or an MPC implementation, as opposed to an integration of the two techniques.

Third, we are working on expanding the language we presented in this paper in a number of ways: (1) by providing support for many other common programming constructs, (2) by allowing the compilation of subsets of existing programming languages to our language, (3) by developing static analysis techniques to aid programmers in reasoning about the MPC portions of their protocols and by developing interfaces that convey analysis results to the programmer, and (4) by exploiting known algebraic properties of the basic operations in the language to optimize the compilation of the high-level program specification into the low-level MapReduce and MPC operations.

Finally, we are also working on supporting alternative assumptions about the privacy constraints imposed on data, whether such data is an input to the analytics or whether such data is derived and used to compute the analytics. For example, our current approach to integrating MPC into MapReduce assumes that all the keys (i.e., the indices used in MapReduce) are public, whereas the values associated with these keys may be private. Such an assumption can be relaxed to leverage currently on-going cryptography research that allows both the keys and the values to be treated as secrets in a key-value store.

Acknowledgements. This work was supported in part by NSF Grants: #1430145, #1414119, #1347522, and #1012798.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters." in *OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, December 2004.

[2] "SCOPE: A Smart-city Cloud-based Open Platform and Ecosystem," https://www.bu.edu/hic/research/scope/, [Accessed: August 15, 2015].

[3] "NCI Cancer Genomics Cloud Pilots," https://cbiit.nci.nih.gov/ncip/nci-cancer-genomics-cloud-pilots/, [Accessed: August 15, 2015].

[4] L. Constantin, "IBM opens up its threat data as part of new security intelligence sharing platform," http://www.infoworld.com/article/2911154/security/ibm-opens-up-its-threat-data-as-part-of-new-security-intelligence-sharing-platform.html, [Online; accessed 15-August-2015].

[5] J. Lipman, "Let's Expose the Gender Pay Gap," http://www.nytimes.com/2015/08/13/opinion/lets-expose-the-gender-pay-gap.html, [Online; accessed 15-August-2015].

[6] R. Barlow, "Computational Thinking Breaks a Logjam," http://www.bu.edu/today/2015/computational-thinking-breaks-a-logjam/, [Online; accessed 15-August-2015].

[7] "ThreatExchange," https://threatexchange.fb.com/, [Online; accessed 15-August-2015].

[8] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[9] Y. Lindell and B. Pinkas, "Secure multiparty computation for privacy-preserving data mining," *Journal of Privacy and Confidentiality*, vol. 1, no. 1, p. 5, 2009.

[10] "VIFF, the Virtual Ideal Functionality Framework," http://viff.dk/, accessed: 2015-08-14.

[11] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A Framework for Fast Privacy-Preserving Computations," in *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08*, ser. Lecture Notes in Computer Science, S. Jajodia and J. Lopez, Eds., vol. 5283. Springer Berlin / Heidelberg, 2008, pp. 192–206.

[12] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *IEEE S & P*, 2015.

[13] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 377–394. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.30

[14] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010, p. 10.

[16] D. Bogdanov, M. Jõemets, S. Siim, and M. Vaht, "How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation," in *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, ser. LNCS. Springer, 2015, to appear.

[17] N. Volgushev, A. Lapets, and A. Bestavros, "Scather: Programming with Multi-party Computation and MapReduce," CS Dept., Boston University, Tech. Rep. BUCS-TR-2015-010, August 2015.

[18] A. Lapets and M. Rötteler, "Abstract Resource Cost Derivation for Logical Quantum Circuit Descriptions," in *Proceedings of the 1st Workshop on Functional Programming Concepts in DSLs (FPCDSL 2013)*, Boston, MA, USA, September 2013.

[19] D. Bogdanov, "How to securely perform computations on secret-shared data," Master's thesis, Tartu University, 2007.

[20] A. Lapets, E. Dunton, K. Holzinger, F. Janssen, and A. Bestavros, "Web-based Multi-Party Computation with Application to Anonymous Aggregate Compensation Analytics," CS Dept., Boston University, Tech. Rep. BUCS-TR-2015-009, August 2015.

[21] K. M. Carter, N. C. Idika, and W. W. Streilein, "Probabilistic threat propagation for network security," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 9, pp. 1394–1405, 2014. [Online]. Available: http://dx.doi.org/10.1109/TIFS.2014.2334272