Scather: Programming with Multi-party Computation and MapReduce

Nikolaj Volgushev Andrei Lapets Azer Bestavros

CS Dept., Boston University
111 Cummington Mall
Boston, MA USA 02215
{nikolaj, lapets, best}@bu.edu

Abstract

We present a prototype of a distributed computational infrastructure, an associated high-level programming language, and an underlying formal framework that allow multiple parties to leverage their own cloud-based computational resources (capable of supporting MapReduce [27] operations) in concert with multi-party computation (MPC) to execute statistical analysis algorithms that have privacy-preserving properties. Our architecture allows a data analyst unfamiliar with MPC to: (1) author an analysis algorithm that is agnostic with regard to data privacy policies, (2) to use an automated process to derive algorithm implementation variants that have different privacy and performance properties, and (3) to compile those implementation variants so that they can be deployed on an infrastructures that allows computations to take place locally within each participant's MapReduce cluster as well as across all the participants' clusters using an MPC protocol. We describe implementation details of the architecture, discuss and demonstrate how the formal framework enables the exploration of tradeoffs between the efficiency and privacy properties of an analysis algorithm, and present two example applications that illustrate how such an infrastructure can be utilized in practice.

1 Introduction

Cloud computing allows multiple institutions, organizations, or agencies to collocate their virtualized resources and data assets, and this in turn enables scalable analytics on such collocated data assets. Two attributes of cloud computing make this possible: (1) the ability to harness significant computational capacities through scalable platforms such as Hadoop [49] and Spark [52] to perform complex computations on the data *in situ*, effectively moving the computation to the data, thus mitigating the risks associated with distributing copies of private data sets, and (2) the opportunity to collocate large data sets owned by multiple organizations on the single physical infrastructure of the cloud provider, thus eliminating the prohibitive costs of moving such data over the network across private infrastructures.

Novel applications and services that leverage very large data sets from a variety of sources can be extremely valuable in a number of settings, with significant payoff to society; these range from scientific discovery [4] to smart-cities [5] and from genomics [3] to homeland and cyber security [23]. Two specific examples that we have targeted in our work are the use of payroll data from multiple institutions to shed light on pay inequities [15, 36, 40] and the use of corporate network data to identify global advanced persistent threats [10].

A major hurdle in truly unleashing the potential of big-data analytics is the justified concern related to proprietary data sets: it is necessary to trust the entity performing computation over the collocated data assets, whether such entity is any one of the data owners or contributors, a third party such as the public cloud provider, or another external entity interested in the result. 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 remains elusive.

1.1 Secure Multi-Party Computation

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 in a way that renders them visible to other entities. 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 being performed. More specifically, secure MPC allows a set of parties to *jointly compute a desirable function* over data they individually hold (the private inputs) while ensuring that the only information that can be gleaned by other parties (or third parties) is the result of the function evaluation and not 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 [48, 51]. Despite significant recent advances [13, 19, 39, 41, 43] in improving the computational efficiency of MPC for computing specific functions and algorithms, as well as in making MPC more accessible by developing libraries and APIs, the impact of MPC remains fairly limited to proof-of-concept studies. This is due to four key challenges. First, the learning curve for using available MPC building blocks is significant, making MPC inaccessible to typical programmers and data analysts. Second, using MPC requires programmers to develop the algorithms they need to deploy from scratch, thus discarding the huge body of existing, time-tested analytics available for popular programming paradigms. Third, existing MPC engines developed to support general purpose computing are standalone solutions; they are not designed to readily leverage scalable cloud platforms in the manner of cloud programming abstractions such as MapReduce [27]. Finally, and perhaps most importantly, for most organizations, the programming/engineering expertise needed to develop scalable analytics (the purview of data analysts) is distinct from the administrative/management expertise needed to assess the privacy implications of using MPC on sensitive data sets (the purview of information security experts and the responsibility of the data contributors themselves). Any realistic solution that aims to leverage MPC for big-data analytics must separate these two concerns and maintain their independence (from the perspectives of these distinct classes of users).

1.2 Integration of Secure MPC into the MapReduce Paradigm

We approach the problem of addressing the accessibility of MPC 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 integrating MPC into the programming paradigm of one of the most popular cloud software development platforms. More specifically, we propose a distributed computational infrastructure and associated high-level programming language that are an extension of MapReduce,[1] arguably the

[1]The MapReduce programming paradigm [27] facilitates processing of large data sets using elastic (virtualized) clusters managed using Hadoop [49] and Apache Spark [52] systems.

most popular cloud-based analytics programming paradigm; our extension introduces programming language constructs that make it possible to leverage MPC in analysis algorithms that access data assets (and, in general, virtualized cloud resources and computational capacity capable of supporting MapReduce operations) that are spread out across (and are under the strict control of) multiple organizations. These new programmaing language constructs make it possible to specify what part of the overall MapReduce computation will need to take place locally within a participating party's MapReduce cluster, across all participants using an MPC protocol, or across all participants with no security guarantees.

Our approach to integrating MPC and MapReduce platforms is informed by (and supports) an important observation: decisions related to constraints on data sharing across institutional boundaries may be orthogonal to the analysis algorithms that need to be computed on that data. Indeed, these constraints may not even be known to the developers of these analysis algorithms. Thus, while the framework we present in this report does allow expert programmers to add MPC constructs to an existing MapReduce algorithm implementation (*i.e.*, an implementation 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), it is also designed to allow this process to be *automated* as a compile-time transformation that can produce secure variants of the implementation coupled with their corresponding security guarantees.[2]

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. Such a language is amenable to static analysis and transformation techniques that can infer (and expose) important tradeoffs between the performance and privacy properties of resulting MPC protocol implementation variants. This furnishes expert programmers with the ability to compare and contrast different design decisions – *e.g.*, the implications of doing various Map and Reduce operations in various orders, or of using different key-value pairs, and so on[3]

1.3 A Formal Framework for Privacy-Performance Tradeoffs

Supporting the execution of statistical analysis algorithms on large collections of data in a distributed setting can be expensive; executing those operations using MPC at that scale can be even more so [14]. While it is possible to work towards improving the performance of distributed computing and MPC primitives and algorithms, an alternative (and complementary) approach is to build an infrastructure and language that expose tradeoffs between the performance (and cost) of the computation needed to execute a particular analysis algorithm and the degree (granularity) of privacy afforded to the data inputs used by the computation. Such an approach can help entities contributing data to quantify the importance of data privacy by tying it to the performance costs of analytics implementations, and it can provide data analysts with flexibilities in accommodating privacy requirements by trading performance for privacy (or vice versa).

For the purposes of this work, we primarily (but not exclusively) consider analysis algorithms with up to three distinct stages: (1) a local computation stage in which the same algorithm portion is executed by each contributing party on its own data, (2) an MPC computation stage in which the

[2]In a real-life deployment scenario, participating parties can then agree on an implementation variant with security guarantees that satisfy their own policies; automating the process of policy reconciliation is an intriguing possibility that we hope to explore in future work.

[3]While this is beyond the scope of this report, assisting programmers with questions related to security and/or correctness of MPC protocols is a separate and important concern that can be addressed with greater formal rigor when working with a uniform language such as the one proposed.

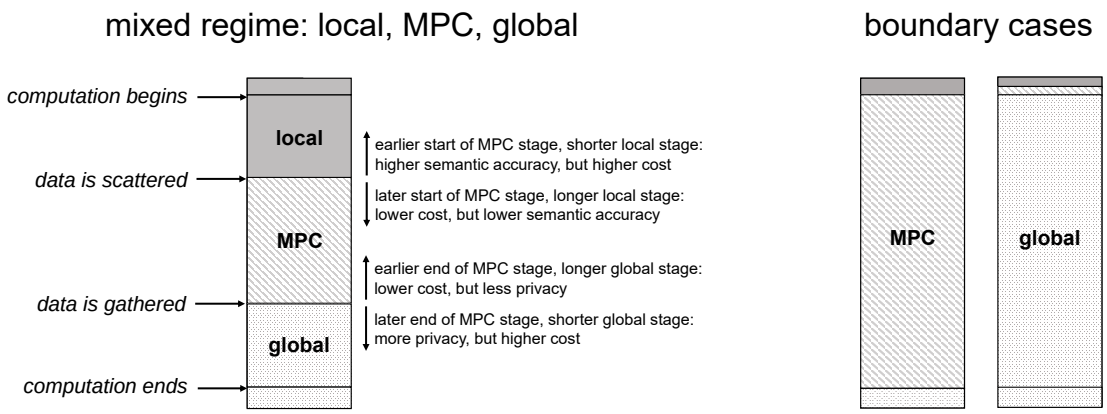

Figure 1: For each data flow in a computation, the points of the transitions from local to MPC computation and from MPC computation to (public) global computation affect cost, accuracy, and privacy.

parties share data securely and collectively perform computations on that data, and (3) a global stage in which computations are computed collectively on distributed data without any guarantee of privacy. This sequence is partly inspired by the stages of a classic MapReduce computation; however, it is possible to conceive of analytics algorithms that switch back and forth between these three stages over the course of their operation. Since MPC protocols may preserve their security guarantees when composed in parallel or sequentially [21], this possibility is worth investigating in future work. Note that in a distributed setting, it is not necessarily the case that all parties (or even all data flows) traverse the three stages in a synchronized way. The most likely synchronization point is the start of the MPC computation, but this is likely to depend on the particular architecture of the platform (and potentially on the policies associated with individual input data sets).

It is possible that for each data flow in an algorithm, the boundaries between each stage can be moved to a point earlier or later in the computation of that data flow (dependency points between data flows necessarily create restrictions on this movement). Movement of the boundary between the local stage and the MPC stage represents a tradeoff between accuracy and performance: on the one hand (a) local computations have no communication overhead, so delaying the transition to the MPC stage as much as possible improves performance and reduces communication costs; on the other hand (b) it may be possible to compromise the accuracy of a computation by delaying the transition even further, *e.g.*, a portion of the computation will simply not incorporate data from other parties, up to some acceptable threshold of divergence in the result. It is also possible that there is some limited tradeoff with regard to privacy, in that the transition to the MPC stage may share some information (for example, the keys in a local key-value store may become public, even if the values remain private).

Movement of the boundary between the MPC computation and the global computation represents a tradeoff between privacy and performance: (a) making the transition to the global computation stage earlier allows the utilization of the distributed computing infrastructure to its full extent, improving performance and reducing costs; (b) delaying the transition to the global computation stage extends the duration of the computation during which data remains private thanks to the guarantees of the MPC protocol being employed. Figure 1 illustrates these stages and boundaries. It also illustrates two possible extremes: if the data over which the analytics algorithm is defined is

shared immediately with all parties, it becomes a simple MapReduce computation. Alternatively, the entire computation could be an instance of MPC. Note that the only situation in which it would be possible for the entire computation to be a local computation is when there is one party (in which case it would be equivalent to a computation that takes place entirely in the global stage).

Many optimization opportunities exist under these conditions. Depending on the mixture of computational resources provided by the participating parties, the mixture of policies associated with the input data, and the particular details of an algorithm, it may be possible to split the work performed by the algorithm under each required regime among groups of participatants in intelligent ways; a methodical exploration of these possibilities can be supported with the creation of an appropriate formal framework and collection of language constructs such as those described in Section 4.

1.4 Organization

The remainder of this report is organized as follows. Section 2 summarizes work related to the framework, language, and infrastructure described in this report. Section 3 describes the architecture and implementation of our integrated backend component. Section 4 defines the high-level programming language and associated formal framework. Section 5 provides two examples illustrating how the overall system can be used, and Section 6 concludes with a summary of our ongoing and future work.

2 Related Work

Work related to the efforts described in this report can be broken down into two categories: (1) research dealing with programming languages and paradigms (and more specifically with cloud programming paradigms and scalable cloud platforms such as MapReduce/Hadoop), and (2) research dealing with MPC protocols (their accessibility to programmers, their application, and their practical performance).

With regard to (1), we are in part interested in creating a formal framework for exploring performance tradeoffs that can arise when defining protocols that utilize cloud programming paradigms. Existing work on the automated static analysis of programs to determine their performance characteristics is currently limited, though some efforts aiming to accomplish this for general programming languages [29, 30, 31] and domain-specific programming languages [37] are under way. The MapReduce programming paradigm is more amenable to such an analysis (compared to a general-purpose programming language) because of its proximity to algebraic and functional programming paradigms, and because many interesting analytics can be constructed with a fairly basic subset of the features it makes available to programmers [33, 46, 47].

While there is a great deal of literature on the broader topics of programming languages and cloud programming paradigms, in this work we are specifically concerned with bringing the prowess of MPC to bear in practical settings; thus, it is most relevant to consider (2) in greater detail.

Prominent theoretical approaches to MPC[4] are linear secret sharing [48] and garbled circuits [51]. Recent efforts aimed at making these approaches accessible to programmers include publicly-available implementations based on linear secret sharing [13, 19, 25, 38] and on garbled circuits [32, 35, 41, 42, 43, 44]. In contrast to our proposed work, these efforts cater to standalone applications in which MPC is implemented in isolation, *i.e.*, without integration into a scalable data processing

[4]For a thorough overview of recent advances in MPC, we refer the reader to the excellent survey by Archer *et al.* [14].

platform and without leveraging existing code bases for large-scale analytics. Also, these standalone solutions (libraries and APIs) presume a significant level of understanding of MPC, which is beyond the reach of typical programmers and data analysts. We brielfy overview some of these efforts below.

Viff [13] was the first MPC framework to be deployed in production. Viff implements an asynchronous protocol for general MPC over arithmetic circuits, based on linear 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 [19], another linear secret sharing scheme, 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, *e.g.*, for financial data analysis [18, 20]. Both Viff and Sharemind are candidate backend frameworks for our language and infrastructure.

A number of recent efforts share our ambition of making MPC more accessible. ObliVM [41] provides a domain-specific language (DSL) for compiling high-level abstractions such as MapReduce operations to oblivious representations. ObliVM implements a garbled circuit backend (but allows for the addition of ther MPC protocols) and supports secure two-party computation. GraphSC [43] extends ObliVM with an API for graph-based algorithms and adds parallelization to the evaluation of the resulting oblivious algorithms. Wysteria [44] provides a DSL which is compiled to a garbled-circuit based MPC encoding. Wysteria allows programmers to explicitly identify segments of code which are to be performed by the parties locally, and segments which require MPC rounds. Both ObliVM and Wysteria offer formal security using type systems. There is follow-up work on using static analysis to infer and improve the performance of MPC protocols [33, 45, 46] along with work on automatic MPC protocol selection and mixed protocol compilation [34, 47]. While these efforts share our ambition of making MPC more accessible, they do not consider MPC as part of a larger, scalable data processing platform, but rather they focus on domain specific uses of MPC. Furthermore, they do not provide the desirable separation of (and opportunity for exposing tradeoffs between) privacy and performance concerns. That said, *many of these advances could be leveraged in support of the back-end of our proposed infrastructure.*

Other efforts aiming to reduce the overhead of MPC have focused on the prohibitive costs of message exchanges (*e.g.*, by restricting communication to subgroups of participants during protocol execution [26, 53]). These efforts are orthogonal to our proposed work as they focus on improving the performance of standalone (or backend) MPC systems.

3 Infrastructure Integrating MapReduce and MPC

Our infrastructure targets a scenario (*e.g.*, a federated cloud [9]) 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 either MapReduce or, more interestingly, an MPC protocol). In particular, each party is equipped with computational resources to (a) execute MapReduce tasks within a private MapReduce cluster, (b) engage with the other participants in the execution of MPC protocols, and (c) coordinate task execution. While our prototype infrastructure utilizes specific implementations of MapReduce and MPC (Apache Spark [52] and Viff [13]), 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 and performance characteristics).

Any of the participating parties (or an external third party interested in some analysis result) can implement their desired analysis algorithm in our high-level programming language (defined in Section 4). The algorithm definition is compiled into a job specification consisting of MapReduce tasks to be executed by each party locally, MPC tasks to be executed across multiple 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 a deployment scenario for our prototype and detail how an analysis algorithm implemented in our high-level language is compiled and executed on the deployed infrastructure.

3.1 Implementation Details

Consistent with the architecture of scalable distributed systems (such as Hadoop and Spark), our prototype infrastructure is comprised of two types of components: *controller nodes*, and *worker nodes*. Each party participating in a protocol execution contributes and controls a worker node (that is therefore trusted by that party). 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.

The controller and worker node software is implemented using Python; this software is available as an Amazon Machine Image (AMI) on Amazon Web Services (AWS). The controller node network interface is a web server implemented in the Python Twisted web framework [11] and serves content over HTTP (we plan to update the controller to serve HTTPS content in the future). All communications between controller and worker nodes are accomplished by sending JSON messages over HTTP (in future versions of the infrastructure, communications will take place over HTTPS). The job specification and all other configuration objects are JSON encodings of Python classes.

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 provides an interface for connecting to an Apache Spark cluster and submitting tasks specified in PySpark, Spark's Python API [6]. The MPC component configures a worker node to participate in the Viff [13] 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 infrastructure 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.

We chose Viff as our backend MPC framework because it implements classical secret sharing schemes and provides an intuitive, lightweight Python API for specifying MPC tasks. Viff's API allowed us to rapidly explore and prototype various designs for our infrastructure and simplified integration. Further, Viff's use of classical secret sharing schemes makes it representative of other MPC frameworks. This aided us in designing an extensible API for the MPC component. Choosing Spark allowed us to explore integration with a state-of-the-art MapReduce platform.

We now concretize the above details by giving an example deployment scenario for our infrastructure.

3.2 Deployment Example

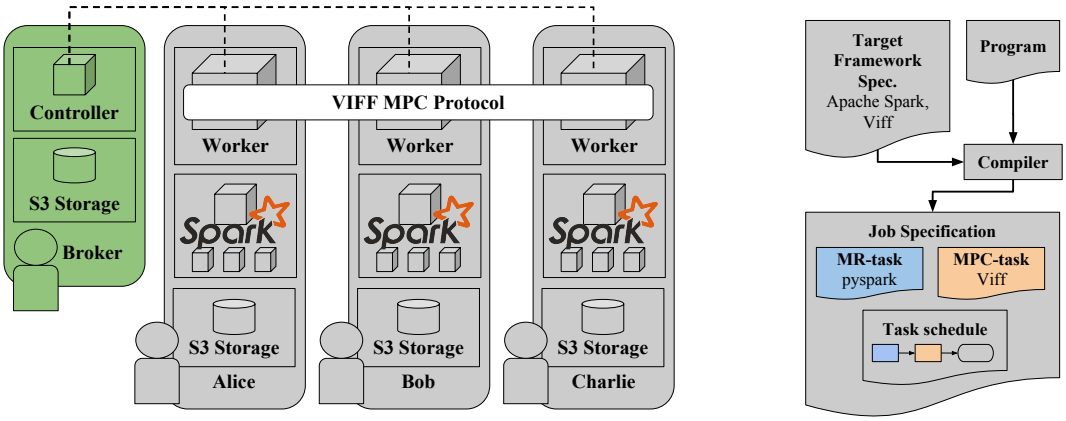

Figure 2: Prototype deployment scenario in a service brokerage setting (left) and compilation workflow of a high-level analysis algorithm definition (right).

Figure 2 (left) illustrates an example prototype deployment. In this scenario, three parties with existing Apache Spark infrastructures and available AWS 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 applications that conform to this scenario in Section 5. 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 AWS 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.[5]

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.

3.3 Compilation of Analysis Algorithms

Analysis algorithms specified in the high-level programming language described in Section 4 can be compiled and executed on a configured infrastructure. Given a high-level algorithm definition 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 2 (right) illustrates an example compilation scenario in which the target frameworks of the compilation are Apache Spark and Viff. The analysis algorithm 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, input, and output operations in Spark's PySpark API. Similarly the MPC task is encoded as a Python module implementing the required operations in Viff.

[5]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.

A job specification produced by the compiler can be executed on a configured infrastructure such as the one described in Section 3.2. We give some concrete examples of high-level analysis algorithms and the resulting task-level encodings in Section 5.

3.4 Execution of Job Specifications

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 delays 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 appropriate private and/or shared key-value stores, performing the specified MapReduce operations on them, and updating the appropriate 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 the integration of Viff into our infrastructure, 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 revealed to the appropriate parties. This can be specified within an analysis algorithm implementation using the **gather** construct, defined in Section 4.1. 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 executing, the final result is reported to the target specified in the algorithm implementation. This could be the controller node, the worker nodes, or a specific subset of worker nodes, depending on the specific application.

4 High-Level Language and Formal Framework

We present a high-level programming language called Scather (in reference to its two built-in constructs **scatter** and **gather**, defined in Section 4.1), along with a proof-of-concept formal framework underlying our proposed infrastructure integrating MapReduce and MPC. A formal framework is

an implementation-independent specification of the requirements that govern the infrastructure's behavior, the semantics of its interfaces, and the mathematical and logical properties that can be derived from these. More importantly, the framework enables the rigorous definition of formal systems that can be the basis for transformations and optimizations of analysis algorithm implementations represented using the language.

4.1 High-level Language for Analysis Algorithm Implementations

Table 1 lists the syntax of our high-level programming language for defining analysis algorithms, where $\mathcal{V}$ is the space of all variables, $\mathcal{L}$ is the space of all string literals, and $\mathcal{T}$ is space of all type names. To illustrate the essential concepts of the framework while minimizing the complexity of the exposition, we introduce a restricted subset of the language.

An algorithm 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 computation that calculates some desired analytic. We use the term *hybrid implementation* or *implementation* to refer to a realization of an analysis algorithm that uses a combination of MapReduce-style and MPC-style computations to produce an analysis result subject to particular privacy constraints.

Expressions in the language are built up using operators that closely resemble those supported by the MapReduce paradigm, as well as a few new operators that bridge the gap between a MapReduce computation and an MPC computation. A *key-value store* is 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). Each *data expression* describes a key-value store in a particular state.[6] In particular, each **store** represents an *initial* key-value store, while every other expression represents an *intermediate* store that exists at some point during the execution of the program. It is assumed that every participant running the protocol has a local key-value store corresponding to each declarations of the form x **:= store(**τ**,** τ**)** (naturally, different participants may have different keys and values within each of these). Each *simple expression* or *expression* describes a key or value.

- The **mapkey** and **mapval** operations take a function and key-value store, and modify every key (given only the key) or every value (given both key and value), respectively, in the key-value store. They are overloaded to also accept a simple expression as the first argument, in which case every key or value in the key-value store specified by the second argument is simply substituted with that simple expression. These restricted variants of **map** are introduced to support policy derivation inference rules such as those presented in Section 4.2, which cannot always be applied when the more general **map** construct is used.
- The **reduce** operation is shorthand for *reduce-by-key*.
- The language is somewhat domain-specific with regard to built-in arithmetic operators, as they are intended to support statistical analyses over (collocated) data sets. Arithmetic operators such as **sum** and **max** can refer to binary operators that accept two integer arguments, to curried operators that take one argument and return a function on integers (*e.g.*, **sum(**1**)** is a function that increments its argument by 1), and to aggregate operators that can be applied using **reduce**.

[6] This key-value store instance could be local to a participant or globally available to all participants. How data available to all is actually distributed and where it is stored is up to the specific compiler and backend implementation.

- The **scatter** and **gather** operations shift the data through the possible operating regimes (the **scatter** operation shifts data from a local to an MPC regime, while the **gather** operation shifts it from an MPC regime to the globally visible regime).[7] 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. These operations can be inserted manually by an expert programmer or automatically at different points of the program (*e.g.*, by the inference rules in Section 4.2).

- Because we assume that keys are never private in our underlying MPC infrastructure, 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(**...**)**).

user type	α	$\in$	$\mathcal{T}$	keys	k	::=	$n \mid \sigma \mid$ **keys(** δ **)**
type	τ	::=	**int** | **bool** | **str** | α	data expression	δ	::=	x
						|	**filter(** k **,** δ **)**
integer	n	$\in$	$\mathbb{Z}$			|	**mapkey(** φ **,** δ **)**
string literal	ℓ	$\in$	$\mathcal{L}$			|	**mapval(** φ **,** δ **)**
constant	c	::=	**true** | **false** | n | ℓ			|	**map(** φ **,** δ **)**
						|	**reduce(** φ **,** δ **)**
variable	x	$\in$	$\mathcal{V}$			|	**union(** δ_1 **,** δ_2 **)**
pattern	p	::=	$x \mid$ **(** p_1 **, ... ,** p_n **)**			|	**update(** δ_1 **,** δ_2 **)**
						|	**join(** δ_1 **,** δ_2 **)**
function	φ	::=	**sum** | **minus** | **prod**			|	**scatter(** δ **)**
		|	**sum(**n**)** | **prod(**n**)**			|	**gather(** δ **)**
		|	**max** | **min**				
		|	**and** | **or** | **not**	statement	s	::=	**type** α **=** τ
		⋮				|	x **:= store(** τ_1 **,** τ_2 **)**
		|	**lambda** p**:** e			|	x **:=** δ
						⋮	
expression	e	::=	x			|	**repeat** n**:** $s_1 \ldots s_n$
		|	c				
		|	**(** e_1 **,** e_2 **)**	program	p	::=	$s\ p$
		|	φ **(** e_1 **,** e_2 **)**			|	**return** δ

Table 1: Abstract syntax for a subset of the high-level programming language for analysis algorithms.

We stipulate that the true (*i.e.*, desired) execution and evaluation semantics of any given analysis algorithm implementation described using the syntax in Table 1 is determined by what would occur if the program were executed on data that is globally visible on a MapReduce infrastructure operated by a single party. For example, the semantics of any implementation derived using the

[7] Those familiar with MPC protocols will appreciate that `scatter` corresponds to the process via which encrypted shares of the private inputs are "scattered" to various parties executing the MPC protocol, whereas `gather` corresponds to the process via which shares representing the output of the MPC computation are "gathered" to yield the results.

process described in Section 4.2 should match the semantics of the policy-agnostic version of the algorithm in which the **scatter** and **gather** operations are absent. We call this the *global semantics*.

In fact, algorithm implementations written using the language can be agnostic to the provenance, location, ownership, and security policies of the data on which they operate; it is only necessary to avoid employing any instances of the **scatter** and **gather** operators in the implementation. This conforms with the abstraction provided by MapReduce [27] (in which the physical distribution of data is abstracted away from the programmer) and can be made compatible with the concept of policy-agnostic programming [50] (in which security and privacy properties are abstracted away from the programmer and specified independently).

4.2 Exploring Tradeoffs through Derivation of Hybrid Implementations

One benefit of having a single high-level language as a front-end for the infrastructure integrating MapReduce and MPC is that it is possible to define static analysis algorithms over protocols that can furnish the programmer with valuable information about the possible tradeoffs the protocol implementation admits between privacy and performance 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.

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 employ MPC may have more flexibility with regard to privacy than with performance, and would find it useful to consider protocol variants that improve performance at the cost of reducing privacy by utilizing MPC in a more restricted way.

We assume that participating parties have individual preferences about how they are willing to share their data. Given an analysis algorithm, the objective is to automatically *derive* an ensemble of possible hybrid implementations of the program that span the range of tradeoffs between privacy and performance. In each implementation in the ensemble, data flows may transition between the local and MPC regimes (as illustrated in Section 1.3, Figure 1) at different points. This ensemble can then be used to enumerate possible security policies for the participating parties (*e.g.*, if one or more parties do not already have a policy, or if their policy is specified using some other representation or level of detail).

$$\textsc{Ident}\ \frac{}{\Omega \vdash \delta \parallel \delta} \qquad \textsc{Var}\ \frac{\Omega(x) = \omega}{\Omega \vdash x \parallel \omega(x)} \qquad \textsc{Union}\ \frac{\Omega \vdash \delta_1 \parallel \omega(\delta_1') \quad \Omega \vdash \delta_2 \parallel \omega(\delta_2')}{\Omega \vdash \mathbf{union}(\delta_1, \delta_2) \parallel \omega(\mathbf{union}(\delta_1', \delta_2'))}$$

$$\textsc{Filter}\ \frac{\Omega \vdash \delta \parallel \omega(\delta')}{\Omega \vdash \mathbf{filter}(k, \delta) \parallel \omega(\mathbf{filter}(k, \delta'))} \qquad \textsc{MapVal}\ \frac{\Omega \vdash \delta \parallel \omega(\delta')}{\Omega \vdash \mathbf{mapval}(\varphi, \delta) \parallel \omega(\mathbf{mapval}(\varphi, \delta'))}$$

$$\textsc{Reduce}\ \frac{\Omega \vdash \delta \parallel \omega(\delta') \quad \omega \text{ and } \varphi \text{ commute}}{\Omega \vdash \mathbf{reduce}(\varphi, \delta) \parallel *\mathbf{reduce}(\varphi, \omega(\mathbf{reduce}(\varphi, \delta')))}$$

$$\textsc{MKR}\ \frac{\Omega \vdash \delta \parallel \omega(\delta') \quad \omega \text{ and } \varphi \text{ commute}}{\Omega \vdash \mathbf{reduce}(\varphi, \mathbf{mapkey}(\ell, \delta)) \parallel *\mathbf{reduce}(\varphi, \omega(\mathbf{reduce}(\varphi, \mathbf{mapkey}(\ell, \delta'))))}$$

Table 2: Selected inference rules for **scatter** obligation propagation over data expressions. The notation $\Omega \vdash \delta \parallel \delta'$ means that under obligation environment Ω, δ can become δ'; ω represents obligations.

$$\textsc{Str}\ \frac{\Omega; \{x \mapsto *\mathbf{scatter}\} \vdash p \parallel q}{\Omega \vdash x := \mathbf{store}(\tau_1, \tau_2); p \parallel x := \mathbf{store}(\tau_1, \tau_2); q} \qquad \textsc{Rtn}\ \frac{\Omega \vdash \delta \parallel \omega(\delta')}{\Omega \vdash \mathbf{return}\ \delta \parallel\ \mathbf{return}\ \omega(\delta')}$$

$$\textsc{Assign-Ident}\ \frac{\Omega \vdash \delta \parallel \delta' \quad \Omega \vdash p \parallel q}{\Omega \vdash x := \delta; p \parallel x := \delta'; q} \qquad \textsc{Assign-Obl}\ \frac{\Omega \vdash \delta \parallel \omega(\delta') \quad \Omega; \{x \mapsto \omega\} \vdash p \parallel q}{\Omega \vdash x := \delta; p \parallel x := \delta'; q}$$

Table 3: Selected inference rules for **scatter** obligation propagation over statements.

Tables 2 and 3 list a collection of derivation rules that can be used to recursively transform an algorithm implementation from one that takes place entirely within an MPC regime into one that takes place partially within a local regime before switching over to an MPC regime. These rules can be viewed as a means for determining the points at which the **scatter** operation can be added to an algorithm definition. The benefit of performing operations locally, if possible, is improved performance. A similar set of rules can be assembled for the transition from the MPC regime to a globally visible regime (where an earlier transition also improves performance at the expense of privacy). Such rules would be a means for determining the points at which the **gather** operation can be added to an implementation.

The inference rules in Tables 2 and 3 work by introducing obligations that force the algorithm implementation to conform to the global semantics (as defined in Section 4.1). An *obligation* is the additional work that must be performed on data in the MPC regime if that data expression is only executed locally within each participant's MapReduce infrastructure on only each participant's local portion of the data. The "initial" obligation is that any key-value store must be immediately scattered, so that every subsequent operation takes place within the MPC regime. The inference rules allow this obligation to gradually be "pushed forward" along each data flow described in the algorithm, assuming certain conditions are met. As an adopted convention, we denote that an operation such as **reduce(sum**, ...**)** is an obligation with the notation ∗**reduce(sum**, ...**)**. In Section 5.1, we illustrate how the inference rules in Tables 2 and 3 can be used to synthesize multiple variants of an algorithm implementation.

5 Example Applications

We present two example applications inspired by the needs of local organizations who require privacy-aware solutions to the mission-critical analysis problems they face: (1) pay-equity analytics over private payroll data from corporate members of of the Boston Women's Compact 100%-Talent initiative [1, 7] and (2) situational awareness analytics over private network data from corporate members of the Mass Insight ACSC threat sharing program [2, 28]. These examples inform the design and development of our framework and demonstrate how our integrated infrastructure can be used to implement a protocol that employs both MapReduce and MPC capabilities. In the remainder of this section, we describe these use cases, elaborating on the first to illustrate some of the considerations that motivated the design decisions and associated research challenges underlying our language and infrastructure.

5.1 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 MPC to tackle important problems with national social justice implications [8, 12, 15]. In particular, the data analysis being considered could identify wage gaps between male and female employees, or can identify the employee seniority level for which the gender wage gap is most extreme.

Implementation Derivation from a Policy-Agnostic Algorithm. Figure 3 shows a simple version of the analysis algorithm, defined using the language syntax presented in Table 1. This algorithm computes the aggregate difference in compensation between female and male employees for a given data store that specifies the gender and salary of each employee.

```
type gender = str
type salary = int
data := store(gender, salary)

f := mapval(prod(-1), reduce(sum, filter("f", data)))
m := reduce(sum, filter("m", data))
d := mapkey("d", union(f, m))

return reduce(sum, d)
```

Figure 3: Policy-agnostic implementation of an algorithm for collecting aggregate compensation data.

Figures 4, 5, 6, and 7 illustrate the data flows for the algorithm in Figure 3; the derivation process defined in Tables 2 and 3 creates (over the course of its operation) a number of hybrid implementations of the initial algorithm implementation. In the diagrams, we omit the second data expression argument in the notation for each operation, as it is implied by the directed arrows representing the data flowing into that operation. Boxed expressions represent operations performed in the local regime, while un-boxed expressions represent operations performed in the MPC regime.

- In the implementation in Figure 4, the entire algorithm executes within the MPC regime.
- The implementation in Figure 5 shows what occurs after the regime boundary shifts to the first **reduce** operation in each flow. This is accomplished using the [Reduce] rule in Table 2 to split each **reduce** operation into a local operation $\mathbf{reduce(sum)}_A$ that only takes place locally on local data on each participating party's MapReduce infrastructure, and an obligation $*\mathbf{reduce(sum)}_{A'}$ to perform it later in the MPC regime on all the participants' data after it has been scattered.
- The shift of the the regime boundary down to the **union** operation via the [Union] rule in Table 2 is shown in Figure 6.
- The shift of the boundary down to the last **reduce** operation via the [MKR] rule in Table 2 is illustrated in Figure 7.

Compilation to Spark and Viff. Figure 8 shows a particular implementation of a variant of the algorithm in Figure 3. In this implementation, all the work to compute the aggregate difference in wages is first computed locally, and then the differences are distributed using secret sharing and aggregated in the MPC regime. Figures 9, and 10 show partial compilation results

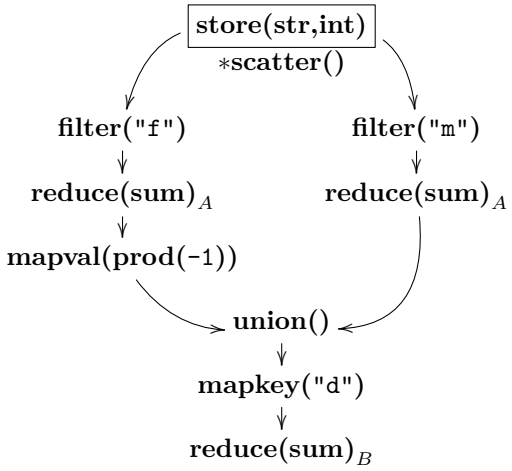

Figure 4: The program is executed entirely in the MPC regime; the only obligation is to immediately **scatter** the initial store as required by the [Str] rule in Table 3.

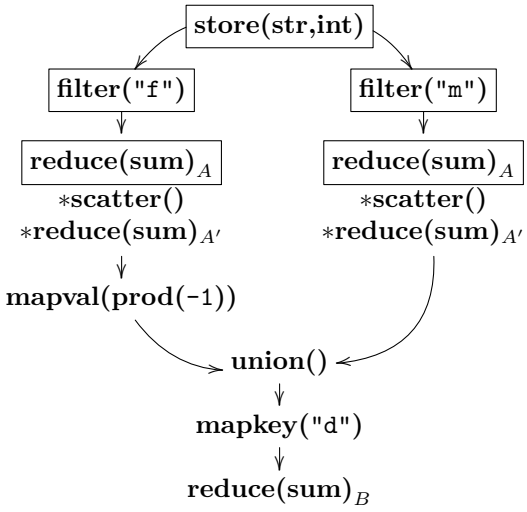

Figure 5: The regime boundary is shifted to the first **reduce** operation in each flow, splitting it into a local operation and an obligation to perform the operation again in the MPC regime.

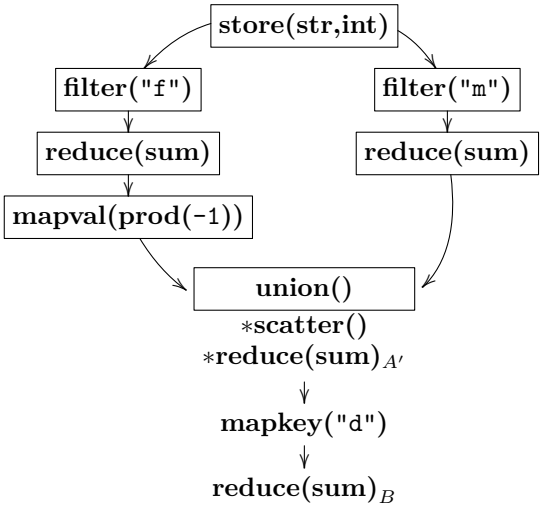

Figure 6: The boundary is shifted down to the **union** operation using the [Union] rule from Table 2, merging the obligations from the two data flows.

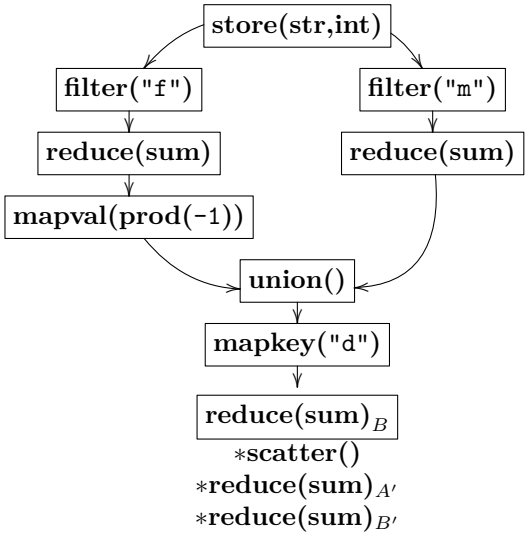

Figure 7: The regime boundary is shifted down to the last **reduce** operation via the [MKR] rule from Table 2. Notice that the **reduce** operation is again split.

for selected sections of the implementation. The compilation of the high-level protocol definition produces distinct tasks. A MapReduce task implements Lines 3–7, followed by an MPC task implementing Line 9. Figure 9 demonstrates a portion of the MapReduce task. Note that a pattern of the form `reduce(sum, union(..., mapval(prod(-1), ...)))` could be recognized by the compiler and transformed into a much simpler expression. A segment of the MPC task is shown in Figure 10. 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.

```
 1: type gender = str
 2: type salary = int
 3: data := store(gender, salary)
 4:
 5: m := reduce(sum, filter("m", data))
 6: f := reduce(sum, filter("f", data))
 7: d := reduce(sum, union(m, mapval(prod(-1), f)))
 8:
 9: s := gather(reduce(sum, scatter(d)))
10: return s
```

Figure 8: Variant implementation for collecting aggregate compensation data (with explicit MPC stage).

```
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)
```

Figure 9: Spark encoding of Lines 3–7 in Figure **??**.

```
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)
```

Figure 10: Viff encoding of Line 9 in Figure **??**.

More Sophisticated Algorithm Variants and Privacy-Performance Trade-offs. Figure 11 shows a more extensive algorithm for computing aggregate compensation data by both gender and seniority. It is more sophisticated than the algorithms in Figures 8 and 11, but like the one in Figure 11 it is privacy-agnostic in that it is written as if the payroll data from the various firms is available in a single data store.

Figure 12 shows a naive implementation variant of the algorithm in Figure 11; this one uses the `scatter` primitive to indicate the need to apply MPC to any manipulation of payroll records in the data store, and uses the `gather` primitive to indicate the point at which it is safe to reveal the results. For purposes of illustration, the `scatter` and `gather` primitives can be seen as a syntactic way to scope the MapReduce code that needs to be performed with MPC protection. As we discussed in Section 4, the `scatter` and `gather` primitives could be introduced into the MapReduce code by an expert programmer or, preferably, by an automated compilation process.

```
type gender = str
type salary = int
type seniority = int
data := store((gender, seniority), salary)

sals_by_gen_sen := reduce(lambda ((gs1, sal1), (gs2, sal2)): sum(sal1, sal2), data)

male_by_sen := mapkey((lambda (gen, sen): sen), filter("male", sals_by_gen_sen))
female_by_sen := mapkey((lambda (gen, sen): sen), filter("female", sals_by_gen_sen))
neg_female_by_sen := mapval(lambda (sen, sal): -sal, female_by_sen)

combined_by_sen := union(male_by_sen, neg_female_by_sen)
pay_gap_by_sen := reduce(sum, combined_by_sen)
pay_gap_by_sen := mapkey('*', mapval(lambda (sen, gap): (sen, gap), pay_gap_by_sen))
max_pay_gap := reduce(lambda (k1, (s1, g1)), (k2, (s2, g2)): max(g1, g2), pay_gap_by_sen)
min_pay_gap := reduce(lambda (k1, (s1, g1)), (k2, (s2, g2)): min(g1, g2), pay_gap_by_sen)

return union(max_pay_gap, min_pay_gap)
```

Figure 11: Policy-agnostic computation of aggregate salary by gender and seniority.

```
type gender = str
type salary = int
type seniority = int
data := scatter(store((gender, seniority), salary))

# ... identical to privacy-agnostic protocol ...

return gather(union(max_pay_gap, min_pay_gap))
```

Figure 12: Private computation of aggregate salary by gender and seniority.

The protocol in Figure 12 requires that all computations are performed as MPC rounds over secret-shared data. This incurs a *significant* degradation in performance [14, 24]. In order to mitigate (at least partially) this performance bottleneck, the MapReduce operations of the original program can be performed locally by each firm, with MPC being employed *only* when the MapRe-

duce operation semantics necessitate aggregation across the results of the local computation. A compilation reflecting this optimization is presented in Figure 13.

```
type gender = str
type salary = int
type seniority = int
data := store((gender, seniority), salary)

sals_by_gen_sen := reduce(lambda ((gs1, sal1), (gs2, sal2)): sum(sal1, sal2), data)

male_by_sen := mapkey((lambda (gen, sen): sen), filter("male", sals_by_gen_sen))
female_by_sen := mapkey((lambda (gen, sen): sen), filter("female", sals_by_gen_sen))
neg_female_by_sen := mapval(lambda (sen, sal): -sal, female_by_sen)

combined_by_sen := union(male_by_sen, neg_female_by_sen)
pay_gap_by_sen := reduce(sum, combined_by_sen)
pay_gap_by_sen := reduce(sum, scatter(pay_gap_by_sen))
pay_gap_by_sen := mapkey('*', mapval(lambda (sen, gap): (sen, gap), pay_gap_by_sen))
max_pay_gap := reduce(lambda (k1, (s1, g1)), (k2, (s2, g2)): max(g1, g2), pay_gap_by_sen)
min_pay_gap := reduce(lambda (k1, (s1, g1)), (k2, (s2, g2)): min(g1, g2), pay_gap_by_sen)

return gather(union(max_pay_gap, min_pay_gap))
```

Figure 13: Optimized privacy-preserving computation of aggregate salary by gender and seniority.

Despite the improved performance of the above approach, the cost of MPC often remains prohibitive when working over large data sets. Even in state-of-the-art implementations [19, 25], certain operations, such as value comparison, require $O(n^2)$ messages to be passed between parties, where n is the number of participants (which can be over 50 in real-world deployments [36]). Performing a `reduce` by `max` operation on a large key-value store thus proves infeasible in MPC. In certain cases, however, it may be possible to move costly operations such as `max` out of the MPC scope by relaxing the privacy constraints on the data.

```
...

combined_by_sen := union(male_by_sen, neg_female_by_sen)
pay_gap_by_sen := reduce(sum, combined_by_sen)
pay_gap_by_sen := reduce(sum, scatter(pay_gap_by_sen))
pay_gap_by_sen := mapkey('*', mapval(lambda (sen, gap): (sen, gap), pay_gap_by_sen))
pay_gap_by_sen := gather(pay_gap_by_sen)
max_pay_gap := reduce(lambda (k1, (s1, g1)), (k2, (s2, g2)): max(g1, g2), pay_gap_by_sen)
min_pay_gap := reduce(lambda (k1, (s1, g1)), (k2, (s2, g2)): min(g1, g2), pay_gap_by_sen)

return union(max_pay_gap, min_pay_gap)
```

Figure 14: Performance-aware privacy-preserving computation of aggregate salary by gender and seniority.

Consider, for example, the alternative compilation shown in Figure 14. The placement of the `gather` operation before computing `max_pay_gap` and `min_pay_gap` allows for the subsequent `reduce` operations to be carried out by the firms' local MapReduce infrastructures. This brings

significant performance gains but requires that the underlying key-value store be made public. In this particular scenario, `pay_gap_by_sen` contains data already aggregated via a `reduce` operation and only reveals the combined pay gaps across seniority levels as opposed to individual firm pay gaps. While it may be in the data analyst's interest to choose the more performant protocol in Figure 14, the firms' data privacy is better protected in the slower protocol in Figure 13. In Section 6.2, we discuss our plans for future work on constructing theoretical and practical tools that can help data contributors and analysts collectively negotiate such trade-offs.

5.2 Collaborative Threat Analysis: Computing Hop Counts to Compromised Nodes

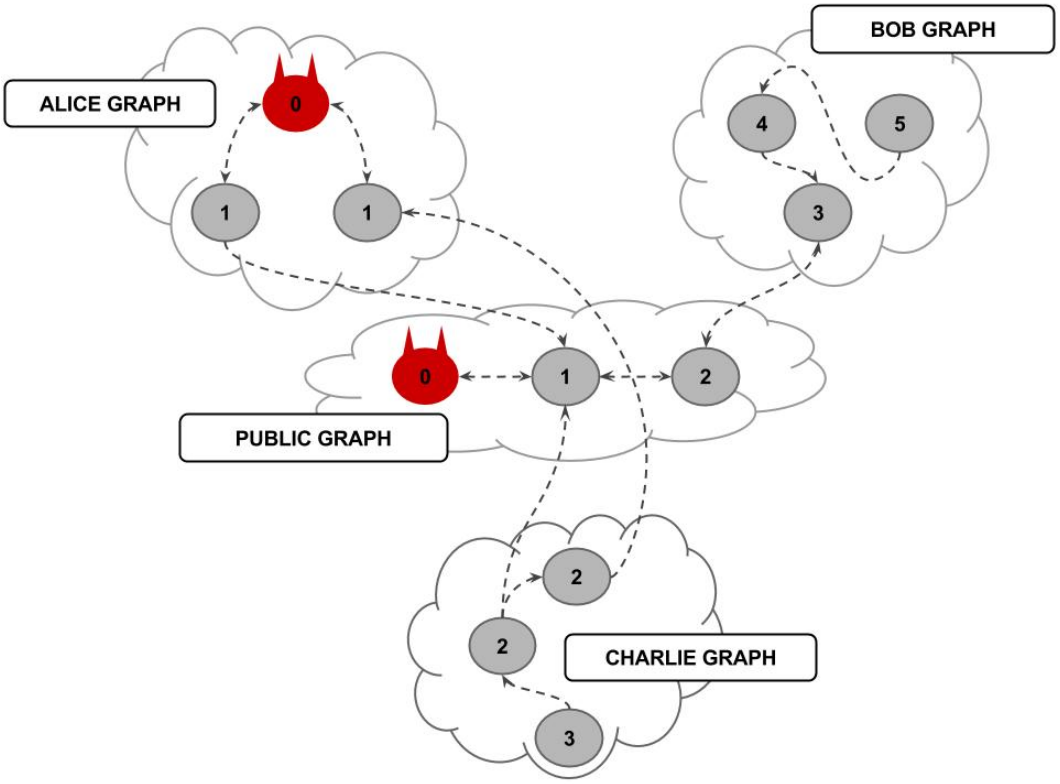

Figure 15: Network with compromised nodes for an instance of the hop count computation use case.

Our second use case concerns the collective analysis of operational data from private networks for situational awareness purposes (*e.g.*, the risk analysis of inter- and intra-network threat propagation [22]). In particular, consider a scenario in which interconnected members of a coalition/consortium of firms [2] (*e.g.*, various banks that each have their own private network) are interested in computing the distance from each of their local network nodes to the closest "potentially compromised" nodes (*e.g.*, running a yet-to-patch kernel). A compromised node may be within the perimeter of another firm. While each firm has an incentive to compute such a risk profile, none of the firms would agree to reveal the "health" of their internal nodes or the "topology" of their internal networks. Here, each party holds private data describing its local network topology, along with metrics about the health of the individual nodes therein. The public interconnection topology (connections between firms through boundary nodes) is known to all participants. As with the compensation statistics use case in Section 5.1, the analysis algorithm to compute the shortest distance (*e.g.*,

hopcount) to a potentially compromised node is straightforward if all the data is available in a single data store. By selectively introducing MPC constructs into the analysis algorithm, one can generate various instances of the analysis algorithm implementation, each exposing private data to a different extent.

```
type node = int
type dist = int

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

cmp_graph := store(node, node)
cmp_dst := store(node, dst)
bdr_dst := store(node, dst)
dst := union(cmp_dst, bdr_dst)

# Begin local computation.
repeat 100:
  nbr_dst := mapval(lambda (v,u): (v,u+1), join(map(lambda x,y: (y,x), cmp_graph), dst))
  dst := reduce(min, union(dst, nbr_dst))

cmp_dst := update(cmp_dst, dst)
bdr_dst := update(bdr_dst, dst)
# End local computation.

# Begin MPC computation.
mpc_dst := scatter(bdr_dist)

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

bdr_dst := gather(filter(keys(bdr_dst), mpc_dst))
# End MPC computation.

dst := union(cmp_dst, bdr_dst)

# Begin local computation.
repeat 100:
  nbr_dst := mapval(lambda (v,u): (v,u+1), join(map(lambda x,y: (y,x), cmp_graph), dst))
  dst := reduce(min, union(dst, nbr_dst))

cmp_dst := update(cmp_dst, dst)
bdr_dst := update(bdr_dst, dst)
# End local computation.
```

Figure 16: Private computation of hop-distance to compromised nodes in a multi-institutional network.

To make this example more concrete, in Figure 16 we provide a simple algorithm implementation using our language. Suppose each party holds private data describing its local network topology, as well as which nodes within the network are compromised. The public network, the boundary nodes (*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.

6 Conclusions and Future Work

In this report 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 analysis algorithms that involves both MapReduce and MPC operations. More importantly, it can provide for programmers a seamless way to compare and contrast the efficiency of different strategies for structuring their implementation of analysis algorithms on private data sets spread out across various administrative domains. Our on-going work is proceeding along a number of different directions.

6.1 Development and Expansion of the Integrated Infrastructure

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 because 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 analysis problem at hand. Therefore, a major thrust of our ongoing work is to extend our coverage of MapReduce and MPC implementations along these lines.

It is possible to generalize 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.

We are also working on supporting alternative assumptions related to 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 such analytics. For example, currently, our 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 are 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.

6.2 Development of the High-level Language and Formal Framework

We are working on further developing the high-level language components we presented in Section 4 in a number of ways: (1) by adding support for other common programming constructs, (2) by allowing the compilation of subsets of existing high-level programming languages for describing analysis algorithms into our language, (3) by exploiting known algebraic properties of the basic operations

in the language to optimize the compilation of the high-level analysis algorithm specification into the low-level MapReduce and MPC operations. Another complementary research direction we are pursuing is the extension of the formal framework (and associated static analysis techniques) to support the automated derivation (directly from analysis algorithm implementations) of security policies and performance characteristics such as accuracy, communication overhead, and running time . We are also interested in developing accessible interfaces that convey the results of these techniques to human users, as these can enable novel opportunities for data contributors and data analysts to negotiate and cooperate in an informed way.

Given an automatically generated ensemble of analysis algorithm implementations and associated policies, it can be useful to report to data contributors the security consequences of each policy as they pertain to the particular data stores they are contributing. Each of the policies derived using a method such as the one described in Section 4.2 can be re-interpreted in a more human-friendly way as a mapping from each data store to the level of disclosure the data will undergo during and after the execution of the analysis algorithm. The level of disclosure can also include any derived information that will be revealed (and whether that information will be derived from individual data or data aggregated across all participants). Catch-all categories can be created for particularly complex levels of disclosure (such categories should necessarily be highly conservative and sound with respect to the privacy guarantees they are reported to imply).

Similarly, a particular analysis algorithm implementation can be automatically converted into an abstract interpretation that predicts the communication overhead and running time of executing the implementation. There are two possible ways this can be accomplished: (1) with concrete, numeric approximations or exact measurements of the size of each initial key-value store (and derived approximate bounds for intermediate stores), or (2) with abstract symbolic formulas (*i.e.*, functions such as polynomials) parameterized by the sizes of the initial key-value stores. The former is more likely to be computable using straightforward techniques when dealing with algorithm implementations that are complex or operate on many initial or intermediate key-value stores. The latter is more general and likely possible for simpler cases or subsets of a language (*e.g.*, a library of common components may pair each component with precomputed symbolic functions describing its communication cost and running time). The latter approach has been employed in work on domain-specific languages for describing quantum circuits [37].

For example, we consider the efficiency of arithmetic operations if Sharemind is used as the MPC backend component [17]. Suppose that the high-level analysis algorithm description contains an expression indicating 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. A set of inference rules over expression syntax trees involving addition and multiplication operations can specify how to recursively assemble a symbolic function describing the communication overhead (parameterized by the number of participants running the MPC protocool). Using these rules, it would be possible to construct a function describing the running time for that particular expression.

One way in which data contributors and data analysts can benefit from an accurate understanding of the tradeoffs that affect the security and performance characteristics of an analysis algorithm implementation is that this understanding can allow them to "negotiate" and choose among analysis algorithm implementations in an informed way. The parties involved in the negotiations can be cognizant of the constraints under consideration (*e.g.*, allowance for performance vs. allowance for privacy leakage), and may even be able to determine and provide adequate compensation, if necessary, in a marketplace setting [16] that monetizes access to private data [**?**]. Supporting such

a marketplace would require the definition of a partial order on data security policies within the formal framework (so that it is possible to determine whether a given policy conforms to the policy specified by a data contributor), as well as user-friendly facilities for associating analysis algorithm implementation variants with corresponding policies.

Acknowledgements

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

References

[1] 100% Talent: The Boston Women's Compact. http://www.cityofboston.gov/women/workforce/compact.asp. [Accessed: August 15, 2015].

[2] ACSC: MassInsight Advanced Cyber Security Center. http://www.acscenter.org/. [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] Open Science Data Cloud. https://www.opensciencedatacloud.org/. [Accessed: August 15, 2015].

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

[6] Spark Python API Docs. http://spark.apache.org/docs/latest/api/python/. [Accessed: August 15, 2015].

[7] The Boston Women Workforce Council. http://www.cityofboston.gov/women/workforce/. [Accessed: August 15, 2015].

[8] The Commonwealth of Massachusetts: Bill H.1733 189th – An Act to establish pay equity. https://malegislature.gov/Bills/189/House/H1733. [Accessed: August 15, 2015].

[9] The Massachusetts Open Cloud Project at Boston University. http://www.bu.edu/moc/. [Accessed: August 15, 2015].

[10] ThreatExchange. https://threatexchange.fb.com/. [Accessed: August 15, 2015].

[11] Twisted. http://twistedmatrix.com/. [Accessed: August 15, 2015].

[12] US Congress Paycheck Fairness Act. https://en.wikipedia.org/wiki/Paycheck_Fairness_Act. [Accessed: August 15, 2015].

[13] VIFF, the Virtual Ideal Functionality Framework. http://viff.dk/. [Accessed: August 15, 2015].

[14] David W Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. 2015.

[15] Rich Barlow. Computational Thinking Breaks a Logjam. http://www.bu.edu/today/2015/computational-thinking-breaks-a-logjam/. [Accessed: August 15, 2015].

[16] Azer Bestavros and Orran Krieger. Towards an open cloud marketplace: Vision and first steps. *IEEE Internet Computing: View from the Cloud*, January 2014.

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

[18] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril 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*, LNCS. Springer, 2015. To appear.

[19] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In Sushil Jajodia and Javier Lopez, editors, *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer Berlin / Heidelberg, 2008.

[20] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis. In AngelosD. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer Berlin Heidelberg, 2012.

[21] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 494–503. ACM, 2002.

[22] Kevin M. Carter, Nwokedi C. Idika, and William W. Streilein. Probabilistic threat propagation for network security. *IEEE Transactions on Information Forensics and Security*, 9(9):1394–1405, 2014.

[23] Lucian 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. [Accessed: August 15, 2015].

[24] Ivan Damgård, Marcel Keller, Enrique Larraia, Christian Miles, and Nigel P Smart. Implementing aes via an actively/covertly secure dishonest-majority mpc protocol. In *Security and Cryptography for Networks*, pages 241–263. Springer, 2012.

[25] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. *Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits*, pages 1–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[26] Varsha Dani, Valerie King, Mahnush Movahedi, and Jared Saia. Quorums quicken queries: Efficient asynchronous secure multiparty computation. In *Distributed Computing and Networking*, pages 242–256. Springer, 2014.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA*, December 2004.

[28] John Dix. New England security group shares threat intelligence, strives to bolster region as cybersecurity mecca, Network World, December 2014. `http://www.networkworld.com/article/2860073/security0/new-england-security-group-shares-threat-intelligence-strives-to-bolster-region-as-cybersecurity-me.html`. [Accessed: August 15, 2015].

[29] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012.

[30] Jan Hoffmann and Zhong Shao. Type-based amortized resource analysis with integers and arrays. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2014.

[31] Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 132–157. Springer, 2015.

[32] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 772–783. ACM, 2012.

[33] Florian Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 703–714. ACM, 2011.

[34] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *Applied Cryptography and Network Security*, pages 566–584. Springer International Publishing, 2014.

[35] Benjamin Kreuter, Abhi Shelat, Benjamin Mood, and Kevin RB Butler. Pcf: A portable circuit format for scalable two-party secure computation. In *Usenix Security*, volume 13, pages 321–336, 2013.

[36] Andrei Lapets, Eric Dunton, Kyle Holzinger, Frederick Jansen, and Azer Bestavros. Web-based Multi-Party Computation with Application to Anonymous Aggregate Compensation Analytics. Technical Report BUCS-TR-2015-009, CS Dept., Boston University, August 2015.

[37] Andrei Lapets and Martin 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.

[38] John Launchbury, Dave Archer, Thomas DuBuisson, and Eric Mertens. Application-scale secure multiparty computation. In *Programming Languages and Systems*, pages 8–26. Springer, 2014.

[39] Yehuda Lindell and Benny Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1(1):5, 2009.

[40] Joanne Lipman. Let's Expose the Gender Pay Gap. `http://www.nytimes.com/2015/08/13/opinion/lets-expose-the-gender-pay-gap.html`. [Accessed: August 15, 2015].

[41] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *IEEE S & P*, 2015.

[42] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA, 2004.

[43] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine 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*, pages 377–394. IEEE Computer Society, 2015.

[44] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 655–670, Washington, DC, USA, 2014. IEEE Computer Society.

[45] Aseem Rastogi, Piotr Mardziel, Michael Hicks, and Matthew A Hammer. Knowledge inference for optimizing secure multi-party computation. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 3–14. ACM, 2013.

[46] Axel Schroepfer and Florian Kerschbaum. Forecasting run-times of secure two-party computation. In *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*, pages 181–190. IEEE, 2011.

[47] Axel Schröpfer, Florian Kerschbaum, and Günter Müller. L1-an intermediate language for mixed-protocol secure computation. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 298–307. IEEE, 2011.

[48] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[50] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. End-to-end policy-agnostic security for database-backed applications. *CoRR*, abs/1507.03513, 2015.

[51] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[52] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

[53] Mahdi Zamani, Mahnush Movahedi, and Jared Saia. Millions of millionaires: Multiparty computation in large networks. *IACR Cryptology ePrint Archive*, 2014:149, 2014.