9-2019

ObliDC: An SGX-based oblivious distributed computing framework with formal proof

Pengfei WU

Qingni SHEN

Robert H. DENG
Singapore Management University, robertdeng@smu.edu.sg

Ximeng LIU

Yinghui ZHANG

See next page for additional authors

Citation

WU, Pengfei; SHEN, Qingni; DENG, Robert H.; LIU, Ximeng; ZHANG, Yinghui; and WU, Zhonghai. ObliDC: An SGX-based oblivious distributed computing framework with formal proof. (2019). *Proceedings of the ACM Asia Conference on Information, Computer and Communications Security (ASIACCS 2019)*. 86-99. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/4512

Author

Pengfei WU, Qingni SHEN, Robert H. DENG, Ximeng LIU, Yinghui ZHANG, and Zhonghai WU

ObliDC: An SGX-based Oblivious Distributed Computing Framework with Formal Proof

Pengfei Wu
Peking University
wpf9808@pku.edu.cn

Qingni Shen*
Peking University
qingnishen@ss.pku.edu.cn

Robert. H. Deng
Singapore Management University
robertdeng@smu.edu.sg

Ximeng Liu
Fuzhou University
snbnix@gmail.com

Yinghui Zhang
Xi'an University of Posts and Telecommunications
yhzhaang@163.com

Zhonghai Wu*
Peking University
wuzh@ss.pku.edu.cn

ABSTRACT

Data privacy is becoming one of the most critical concerns in cloud computing. Several proposals based on Intel SGX such as VC3 [1] and M^2R [2] have been introduced in the literature to protect data privacy during job execution in the cloud. However, a comprehensive formal proof of their security guarantees is still lacking. In this paper, we propose ObliDC, a general UC-secure SGX-based oblivious distributed computing framework. First, we model the life-cycle of a distributed computing job as data-flow graphs. Under the assumption of malicious, adaptive adversaries in the cloud, we then formally define data privacy of a distributed computing job by introducing a notion named ODC-PRIVACY, which encompasses both semantic security (to protect data confidentiality during computation and transmission) and oblivious traffic (to prevent data leakage from traffic analysis). ObliDC is composed of four two-party protocols — job deployment, job initialization, job execution, and results return, which allow for modular construction of concrete privacy-preserving job protocols in different distributed computing frameworks. Finally, inspired by a formal abstraction for trusted processors proposed by R. Pass et al. [3], we formally prove the security of ObliDC under the universal composability (UC) framework.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**.

KEYWORDS

oblivious computation, Intel SGX, distributed computing systems, formal proof

ACM Reference Format:
Pengfei Wu, Qingni Shen, Robert. H. Deng, Ximeng Liu, Yinghui Zhang, and Zhonghai Wu. 2019. ObliDC: An SGX-based Oblivious Distributed Computing Framework with Formal Proof. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3321705.3329822

1 INTRODUCTION

Cloud computing is the most popular platform for distributed computing nowadays. Users can deploy various distributed computing frameworks, e.g., MapReduce [4], Spark [5] and Storm [6], by renting virtual and scalable resources from cloud service providers. It is unsurprising that cloud computing has drawn much attention from the governments to industries. For example, the cloud computing market of the U.S Federal Government has grown at a rate of 16.2% annually from 2015-2020 and will grow up to $10 billion by 2020 [7]. Cisco forecasts that the cloud network traffic will grow from 3.9ZB in 2015 to more than 14.1ZB in 2020 [8]. Moreover, some research institutions have been set up to devote to cloud computing research, such as [9] and [10].

Distributed computing frameworks, deployed in a master-slave architecture, are most commonly used in processing massive amount of data in the cloud. A distributed *job* can be usually split into different *tasks* in *phases*. Upon a *user* providing a job request and tasks being scheduled, networked *servers* (also known as *worker nodes*) work cooperatively on this job. However, job-dependent data provided by the user can be highly sensitive, and a curious cloud service provider or a third party may try to obtain or infer user's privacy as follows: (1) *Data exposure during task execution.* After a task being scheduled in a server, the server works on the input data and generates task results. Even if the input data is in ciphertext form to protect confidentiality over the network (i.e., encrypted in AES [11] or RSA [12]), it has to be decrypted before processing, and the decrypted data will be exposed to the server directly. (2) *Privacy inference through traffic analysis.* Previous work shows that although the content of data is encrypted, the *access pattern* of data traffic can still leak sensitive information from user's data to the adversary [13]. For example, in the shuffle process of MapReduce, an adversary can infer whether two key-value pairs have the same key by observing the data traffic between a mapper and a reducer [2]. What's worse, if the adversary has some background knowledge about the input data, it may obtain a user's data in plaintext by statistical inference [14].

One approach to protect data privacy in task execution is using *homomorphic encryption* [15], which allows direct computation on

*Both are corresponding authors.

ASIACCS '19, July 9–12, 2019, Auckland, New Zealand
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329822

ciphertexts. During task execution, the cloud is unable to learn the underlying plaintext since data stays in encrypted form throughout the entire task execution in servers. However, homomorphic encryption has serious limitations. Though *fully homomorphic encryption* can handle arbitrary computations, it suffers from extremely high computational and storage overhead, while *semi homomorphic encryption* incurs much lower computational complexity, it can only perform very limited computations [16].

Another approach for realizing secure computation is using trusted computing technology such as Software Guard Extensions in Intel processors (Intel SGX) [1],[2],[14],[17],[18], which allows a program to be executed in a secure manner such that the data and inner states of execution cannot be observed and tampered with directly. The SGX-enabled processor creates an isolated area in memory, called *enclave*, which cannot be accessed from the operating system and hypervisor. SGX also provides a mechanism, named *remote attestation*, to enable users to verify whether the code has been loaded into the enclave securely and correctly. Compared with homomorphic encryption, there are two apparent advantages for realizing secure computation with Intel SGX processor: (1) high performance. Previous efforts have validated that no remarkable difference in performance between privacy preserving job execution in SGX-enabled processors and normal job execution without any privacy protection [1]; (2) arbitrary tasks supported. Users only need to define the sensitive code to be run in an enclave in C language to support arbitrary computations.

Among existing SGX-based solutions, VC3 [1] is a secure distributed computing framework for MapReduce [4], which protects data confidentiality by executing map and reduce tasks in enclaves and by encrypting data transmitted over the network using an *IND-CPA* encryption scheme [19]. Recent studies [2],[14] find that VC3 is vulnerable to traffic analysis, wherein data privacy can be leaked from the network-level access pattern. However, all these existing efforts only adopt heuristic proof and semi-formal reasoning in their security analysis [3], and a rigorous formal security analysis of privacy-preserving distributed computing remains an open problem.

1.1 Our Contributions

Our goal is to bridge the gap between the design and provable security in SGX-based secure distributed computing protocol by presenting an **Obli**vious **D**istributed **C**omputing (ObliDC) framework which follows the paradigm of MapReduce [4] and Spark [5]. Main contributions of the paper are summarized below.

• *Modular Construction of Job Protocols*: Given that a job process in distributed computing framework can be affected by many factors including *programming paradigm* [20] and system parameters (i.e., the input size and file block size), we need a general model which hides the underlying factors in order to facilitate our generic design and analysis. Hence, in ObliDC, we model the life-cycle of a job as directed *job data-flow graphs* (in Section 4.1) and decompose the life-cycle of a job in a distributed framework into four phases: (1) *Job Deployment*, a user sets up a secret key with servers and then submits job codes; (2) *Job Initialization*, the user uploads input data and servers read them from the distributed storage system, then work on it; (3) *Job Execution*, servers perform job codes and transfer intermediate results with each other; and (4) *Results Return*, upon completion of the tasks, the user downloads the job outputs from the servers. Formally, ObliDC consists of four fined-grained two-party protocols, which cover the four phases of the life-cycle of jobs in distributed computing frameworks. Any concrete job protocols can be instantiated based on these basic protocols.

• *Resistant to Traffic Analysis*: Combined with the graph model, we identify privacy leakage due to the network-level access patterns. Taking semantically secure encryption into consideration as well, we summarize two security properties required in job protocols in an indistinguishability-based security notion named **O**blivious **D**istributed **C**omputation - privacy (ODC-PRIVACY) (in Section 4.3). Specifically, ODC-PRIVACY encompasses both *semantic security* to protect data confidentiality during computation and transmission and *oblivious traffic* to prevent privacy leakage from traffic analysis. ObliDC is designed to provide ODC-PRIVACY and this is achieved by proposing two privacy-enhancing preconditions: *oblivious traffic direction* (by enabling the same number of data blocks to be transferred between tasks during shuffle) and *oblivious traffic size* (by padding the size of each transferred block to the same length) (in Section 4.2). Any concrete job protocols as instances of ObliDC will inherit its security property and be able to provide semantic security and oblivious traffic.

• *Formal Description of Code Execution in Intel SGX*: ObliDC splits a job code into the *sensitive* code and the *non-sensitive* code. The former is specific for a job while the latter is code shared among various jobs for common functionalities such as key generation, encryption, and decryption. Code execution process in SGX enclave is formally defined and described in ObliDC, including code installation, remote attestation, and function invocation (see Section 5 for details).

• *Formal Security Analysis in UC Framework*: Previous work demonstrates that data confidentiality in distributed computing frameworks can be proved in games [21]. However, this proof technique is only suitable for a complete and specific protocol and is hard to be adapted to our modular and genetic framework. Hence, in ObliDC, we follow the *universal composability* (UC) framework [22],[23], a simulation-based proof technique, in security proof. Inspired by a formal abstraction for Intel SGX proposed by R. Pass et al. [3], we take an SGX-enabled processor as a black-box $\mathcal{G}_{att}$. To prove the security of job protocols in the UC framework, we first provide the ideal functionality for the oblivious distributed computing and demonstrate the security equivalence between the ideal functionality and ODC-PRIVACY (in Section 6.1). Then, we propose the ideal functionality for each subroutine in ObliDC framework and prove their security in the UC framework, respectively (in Section 6.2). At last, we show a protocol instance of our framework can be UC-secure in the hybrid model of these four ideal functionalities.

• *Formal Analysis of VC3 and M^2R*: We apply ObliDC to formally analyze VC3 [1] and M^2R [2]. We show why a job in a secure-shuffled VC3 and M^2R can be secure in the sense of ODC-PRIVACY. To keep the paper compact, we leave the analysis in Appendix D.

1.2 Organization

The rest of the paper is organized as follows. In Section 2, we introduce some necessary background knowledge. In Section 3, we

Table 1: Summary of Notations

Notations	Descriptions
$\mathcal{G}_{att}$	a formal abstraction of a secure processor
$c = \mathsf{Enc}\{k, m\}$	encryption of plaintext m with key k
$m = \mathsf{Dec}\{k, c\}$	decryption of ciphertext c with key k
$\mathbb{G} = (V, E)$	a directed graph $\mathbb{G}$ with vertex set V and edge set E
$\mathbb{G}_s(X)(\mathbb{G}_s(Y))$	the disjoint subset partitioned from the vertex set in a complete bipartite graph $\mathbb{G}_s$
$\ell(v_i, v_j)$	a directed edge from v_i to v_j
$d^+_{\mathbb{G}}(v)$ $(d^-_{\mathbb{G}}(v))$	the inner (outer) degree of vertex v in graph $\mathbb{G}$
C	a constant for the traffic size

define the system model and the adversary model. In Section 4, we discuss the job data-flow graphs model, propose preconditions against traffic analysis and define ODC-PRIVACY. In Section 5, we introduce the detailed design of the ObliDC framework. We provide formal security proof of ObliDC in the UC framework in Section 6 and evaluate the performances of ObliDC in Section 7. We survey related work in Section 8. Finally, we conclude and discuss possible future research directions in Section 9.

2 PRELIMINARY

In Table 1, we summarize some helpful notations used in ObliDC.

2.1 Intel SGX

Intel SGX is a CPU extension scheme for executing code security in Intel processors [24]. To prevent from accessing and tampering with other applications by operating system and hypervisor, Intel SGX creates an isolated area in memory, named enclave. All messages transmitted between CPU cache and the enclave are in an encrypted form. Hence, the trusted computing base (TCB) only contains the processor itself and programs defined in the enclave. After the code having been submitted, a user can verify whether the code is loaded securely and correctly by remote attestation. A secure channel is then established between the user and enclave to protect their subsequent communications.

When programming enclave applications, a user can define his sensitive code in the `.edl` file. The running application is divided into two parts: the *trusted* and the *untrusted*. The routine outside the enclave invokes the function inside is called `ECALL` and the function inside invokes the routine outside is called `OCALL`. For a more detailed description about Intel SGX, the reader is referred to [25],[26].

2.2 Universal Composability

The universal composability (UC) framework allows for modular security analysis of complicated protocols [22]. Subroutines of a protocol can be analyzed separately. UC ensures that a protocol composed of UC-secure subroutines is UC-secure as well. Generally speaking, a complicated protocol ρ calls an ideal functionality $\mathcal{F}$, and the security properties of ρ will be retained, if replacing $\mathcal{F}$ by an actual sub-protocol π realizing it. In the UC framework, a protocol is represented as a system of probabilistic Interactive Turing Machines (ITMs) [22],[23], where each ITM represents the program to be run within a protocol party $\mathcal{P}$. The process of executing a protocol in the presence of an adversary $\mathcal{A}$ is called the *real world*. The *ideal world* defines an ideal functionality $\mathcal{F}$, which plays a role of the "trusted party", and a *simulator* $\mathcal{S}$, which operates by simulating an execution of $\mathcal{A}$. All protocol participants in the ideal world cannot communicate with each other. The environment outside the given protocol is defined by an environment machine $\mathcal{Z}$, which provides protocol inputs to all parties and sees their outputs. We say that a protocol π UC-realizes $\mathcal{F}$, if for any probabilistic polynomial time (*PPT*) adversaries $\mathcal{A}$, there exists a *PPT* simulator $\mathcal{S}$, such that none *PPT* environment $\mathcal{Z}$ is able to tell whether it contacts with π and $\mathcal{A}$ in the real world or $\mathcal{S}$ in the ideal world. If we use $\overset{c}{\equiv}$ to denote the *computational indistinguishability*, we then have

$$\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \overset{c}{\equiv} \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$$

```
$\mathcal{G}_{att}[\Sigma,\mathsf{reg}]$
init(): //$initialization$
    $(mpk, msk)$:=$\Sigma$.KeyGen($1^\lambda$)
    $T = \emptyset$
getpk() from $\mathcal{P}$: //$public\ interface$
    return $mpk$ to $\mathcal{P}$
install($idx$, prog) from $\mathcal{P} \in$ reg: //$local\ interface - installation$
    if $\mathcal{P}$ is honest, assert $idx = sid$
    generate $eid \in \{0,1\}^\lambda$
    $T[eid, \mathcal{P}] := (idx, \text{prog}, \vec{0})$
    return $eid$ to $\mathcal{P}$
resume($eid$, $inp$) from $\mathcal{P} \in$ reg: //$local\ interface - resume$
    $(idx, \text{prog}, mem) := T[eid, \mathcal{P}]$, abort if not found
    $(outp, mem) := \text{prog}(inp, mem)$
    $T[eid, \mathcal{P}] := (idx, \text{prog}, mem)$
    $\sigma := \Sigma.\text{Sig}_{msk}(idx, eid, \text{prog}, outp)$
    return $(outp, \sigma)$ to $\mathcal{P}$
```

Figure 1: The formal abstraction of secure processors $\mathcal{G}_{att}$

2.3 Formal Abstraction for Secure Processors

R. Pass et al. [3] proposed a formal model, denoted by $\mathcal{G}_{att}$, for attested execution secure processors. This allows one to study high-level cryptographic protocol design based on secure processors. Under the assumption that more than one program is executed in one processor, $\mathcal{G}_{att}$ is modeled as a global shared trusted setup functionality in the Generalized UC (GUC) model [27], which provides a formal abstraction for common trusted processors (see Fig. 1), including

• **Initialization**. Upon initializing a processor, the manufacturer $\mathcal{M}$ selects a signature scheme Σ, which parameterizes a pair of secret keys (*mpk*, *msk*) for the processor. The private key *msk* is kept in the enclave and used to sign an attestation. While the public key *mpk* can be revealed to prove the identity. After parameterization, $\mathcal{M}$ empties a data structure T, which is used for recording enclave information.

• **Registration**. During registration of a new machine $\mathcal{P}$ equipped with a trusted processor, the registry `reg` records $\mathcal{P}$'s identity and some other information (i.e., *mpk* of the processor in $\mathcal{P}$). Machines only in the list of `reg` can call enclave programs and produce attestations using *msk*.

• **Public Interface**. $\mathcal{G}_{att}$ allows any parties to send requests for the public key *mpk* of the trusted processor. This models the process that a processor distributes its public key to arbitrary parties during attestation verification.

• **Local Interface**. Local interfaces model interactions of the trusted processor with the local host machine $\mathcal{P}$. (1) `install`. If $\mathcal{P}$ sends an `install` instruction to the $\mathcal{G}_{att}$, a new enclave is established binding with some identifier *idx*. If $\mathcal{P}$ is honest, *idx* is identical to the session identifier of the current protocol provided by $\mathcal{P}$. $\mathcal{G}_{att}$ then generates a unique enclave identifier *eid*, and stores it with the program prog into the enclave together. Finally, $\mathcal{G}_{att}$ returns *eid* to caller $\mathcal{P}$. (2) `resume`. Upon receiving `resume` instruction, $\mathcal{G}_{att}$ checks whether the enclave ID *eid* exists in data structure T. It will abort the protocol if *eid* is not found. $\mathcal{G}_{att}$ then executes the program prog on user's input *inp*, and updates the inner state of the memory *mem*. The attestation σ is produced by signing the output *outp* and prog using *msk* under the signature scheme Σ. Later, *outp* and σ are returned to the caller $\mathcal{P}$.

3 SYSTEM SETUP AND ADVERSARY MODEL

3.1 System Setup

In ObliDC, we mainly focus on how a distributed job is performed in a privacy-preserving manner. As depicted in Fig. 2(a), the system comprises a user and a distributed computing framework consisting of multi servers:

• *User* $\mathcal{U}$: The user is responsible for sending a job request and provides inputs for servers. Before job execution, a user encrypts the job code with a temporary key and distributes the key to the enclave of each server over the secure channel established by the remote attestation. After that, $\mathcal{U}$ encrypts the job inputs with the temporary key again and delivers them to servers performing tasks. After the job terminates, $\mathcal{U}$ downloads the results from the distributed computing framework and decrypts the results with the temporary key.

• *Distributed Computing Framework*: The distributed computing framework is deployed on a cluster of servers which are network-connected in a specific topology. In ObliDC, each server $\mathcal{P}$ is equipped with an SGX-enabled processor for securely performing a task. Moreover, $\mathcal{P}$ is allowed to store some job-dependent data in local storage (i.e., disk or memory), including task inputs and intermediate results. After a task being scheduled, $\mathcal{P}$ installs user-provided code into the enclave and decrypts task inputs with the temporary key. Before the task results leave the enclave, they will be encrypted again and sent to other servers for further processing[1].

3.2 Adversary Model

We consider a *malicious* and *adaptive* insider adversary $\mathcal{A}$ (i.e., cloud service provider or cloud administrator), which can arbitrarily deviate from the protocol specification to learn user's sensitive data during job execution [28]. Moreover, $\mathcal{A}$ can be adaptive in the sense that it can corrupt *any* servers at *any* time. The choice of whom and when to corrupt can be arbitrarily decided by $\mathcal{A}$, and it depends on its view of execution [28]. $\mathcal{A}$ can monitor the network communication between a user and a server, as well as between servers. After $\mathcal{A}$ corrupting a server $\mathcal{P}$, any information in software stack outside the TCB can be accessed directly, including data on disk or in the memory. The adversary can launch several passive attacks to infer user's privacy, such as ciphertext analysis to extract input data in plaintext and traffic analysis by observing whether two key-value pairs are shuffled to the same task [2]. What's worse, if $\mathcal{A}$ has some background knowledge about the job input such as data distribution or frequency, it can infer more sensitive information. The adversary can perform correlation analysis on what it observes and the background knowledge. Previous work has shown that $\mathcal{A}$ can successfully infer some individual attributes from a census database using background knowledge [14]. For the active attacks, we allow the adversary to abort the job protocol at any time. Firstly, in MapReduce, the adversary can simply drop the output of a mapper to count the value of the final output reduced, which exposes the output of this dropped mapper. Secondly, the adversary also can reorder or misroute intermediate results of a mapper from an intended reducer to another, which will leak the outputs of this mapper as well [2].

In ObliDC, we assume that the user is honest, and all tasks are performed inside the SGX enclave as in VC3 [1] and M^2R [2]. Moreover, the adversary $\mathcal{A}$ cannot break into the TCB, as well as performing a denial-of-service attack, hardware attack and other side-channel attacks (i.e., time-channel attack [2] or from memory-level access pattern [29]) on SGX. Concretely, we consider the provable security of the following two requirements in ObliDC:

(1) *Confidentiality:* All inputs and intermediate data are in encrypted form during transmission over the network. The adversary is unable to extract the plaintext of these data in polynomial time.
(2) *Oblivious Traffic:* The malicious adversary is unable to infer user's sensitive data by observing traffic pattern or taking some active attacks (i.e., dropping tuples or misrouting data blocks as above).

4 FORMAL COMPUTING MODEL AND DEFINITION OF DATA PRIVACY

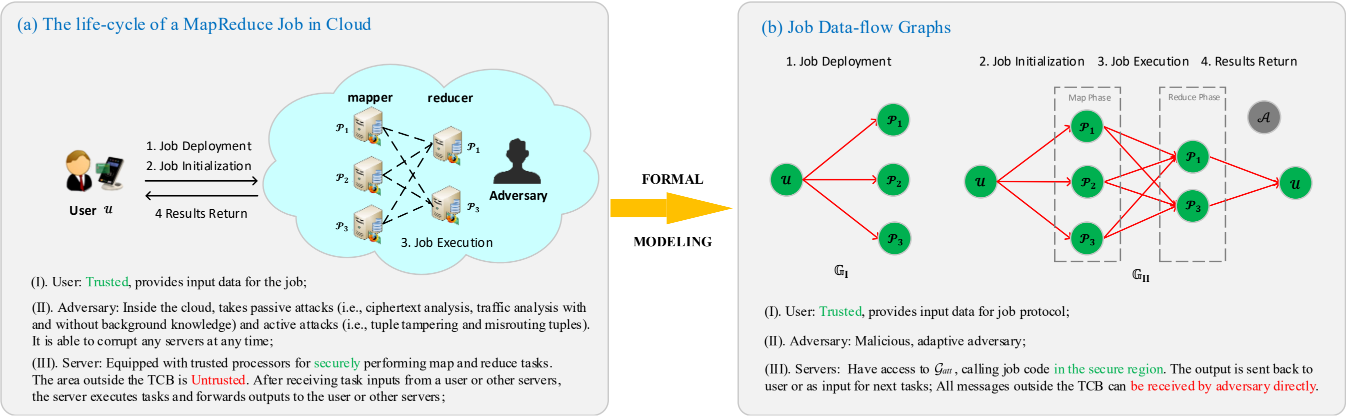
In this section, we first formalize the definition of the life-cycle of a job in a distributed computing framework (Section 4.1). Before formally define the data privacy ODC-PRIVACY, we introduce concepts of oblivious traffic direction and oblivious traffic size as two necessary preconditions for oblivious traffic against passive and active traffic analyses mentioned in Section 3.2.

4.1 Formal Computing Model

In ObliDC, We model the life-cycle of a job in distributed computing framework by a pair of directed graphs $\mathbb{G}_{\mathrm{I}}$ (for job deployment) and $\mathbb{G}_{\mathrm{II}}$ (for job initialization, job execution and results return) as job data-flow graphs based on observing a data flow exists between two parties (including the user and servers). The graph model is defined as follows:

Definition 1 (Job Data-flow Graphs). *We assume that there are n servers participating in a job execution. The life-cycle of a job*

[1] Especially, if two tasks are continuously performing in the same server, a part of intermediate results generated by the parent task will then be kept as the input for the child task without being sent out.



(I). User: Trusted, provides input data for the job;

(II). Adversary: Inside the cloud, takes passive attacks (i.e., ciphertext analysis, traffic analysis with and without background knowledge) and active attacks (i.e., tuple tampering and misrouting tuples). It is able to corrupt any servers at any time;

(III). Server: Equipped with trusted processors for securely performing map and reduce tasks. The area outside the TCB is Untrusted. After receiving task inputs from a user or other servers, the server executes tasks and forwards outputs to the user or other servers;

(I). User: Trusted, provides input data for job protocol;

(II). Adversary: Malicious, adaptive adversary;

(III). Servers: Have access to $\mathcal{G}_{att}$, calling job code in the secure region. The output is sent back to user or as input for next tasks; All messages outside the TCB can be received by adversary directly.

Figure 2: An overview of the formal model for the life-cycle of a MapReduce job as job data-flow graphs

in distributed computing framework forms a two-tuple of directed graphs $\mathbb{G} = \{\mathbb{G}_{\mathrm{I}}, \mathbb{G}_{\mathrm{II}}\}$:

- *$\mathbb{G}_{\mathrm{I}} = (V_{\mathrm{I}}, E_{\mathrm{I}})$, where V_{I} is a set of the user $\mathcal{U}$ and the enclave on each server $\mathcal{P}_i, i \in \{1, 2, \ldots, n\}$. A directed edge $\ell(\mathcal{U}, \mathcal{P}_i) \in E_{\mathrm{I}}$ denotes a user securely delivers job codes to $\mathcal{P}_i$'s enclave.*
- *$\mathbb{G}_{\mathrm{II}} = (V_{\mathrm{II}}, E_{\mathrm{II}})$, where V_{II} is a set of the user $\mathcal{U}$ and the task performed on $\mathcal{P}_i, i \in \{1, 2, \ldots, n\}$. A directed edge $\ell(v_i, v_j) \in E_{\mathrm{II}}$ represents job-dependent data (i.e., job inputs, intermediate results and job outputs) are transferred from vertex v_i to v_j. Both source and sink points of $\mathbb{G}_{\mathrm{II}}$ are user $\mathcal{U}$, which imply the user uploads inputs and downloads outputs from the distributed computing framework.*

Fig. 2(b) shows an example of job data-flow graphs $\mathbb{G} = \{\mathbb{G}_{\mathrm{I}}, \mathbb{G}_{\mathrm{II}}\}$ formally modeling a real-world MapReduce job involving three servers. Before performing the job, a user submits codes to enclaves of these three servers as depicted in $\mathbb{G}_{\mathrm{I}}$. The transmission of job-dependent data during job initialization (from the user to three mappers), job execution (from three mappers to two reducers) and results return (from two reducers to the user) are depicted in $\mathbb{G}_{\mathrm{II}}$. We remark that in the job data-flow graphs, data is stored in either the distributed storage system or local storage before loading into enclaves, we combine this data storage/retrieval process with data transmission and denote them as an edge in the graph.

4.2 Preconditions Against Traffic Analysis

In Section 3.2, we have discussed the privacy leakage due to traffic analysis with and without background knowledge as well as some active attacks. We now propose two preconditions based on the job data-flow graphs $\mathbb{G}$, namely oblivious traffic direction and oblivious traffic size, to thwart these traffic analyses.

Oblivious Traffic Direction. During the shuffle process in a distributed computing framework, tasks in the parent phase generate a series of key-value pairs $\langle key, value \rangle$. The results with the same *key* are shuffled to the same child task for further processing. Due to the fact in some jobs, some tasks in the parent phase may have no appropriate key-value pairs as the inputs for some specific child tasks. An adversary may infer whether some tasks performed in the parent phase generate intermediate results with the same *key*. Hence, as in some privacy-preserving solutions like Melbourne shuffle [14] and cascaded mix network [2], tasks in the parent phase have to send output blocks to *all* tasks in the child phase. In this way, it makes the traffic direction oblivious.

Definition 2 (Oblivious Traffic Direction). *In the directed graph $\mathbb{G}_{\mathrm{II}} = (V_{\mathrm{II}}, E_{\mathrm{II}})$ of job data-flow graphs, the shuffle process forms a directed subgraph $\mathbb{G}_s = (V_s, E_s), \mathbb{G}_s \subseteq \mathbb{G}_{\mathrm{II}}$, where $V_s = \mathbb{G}_s(X) \bigcup \mathbb{G}_s(Y)$. If the traffic direction is oblivious, for all $v_i \in X, v_j \in Y$, $\ell(v_i, v_j) \in E_s$, we then have $\forall v_i \in \mathbb{G}_s(X)$, $d^-_{\mathbb{G}_s}(v_i) = |\mathbb{G}_s(Y)|$ as well as $\forall v_j \in Y$, $d^+_{\mathbb{G}_s}(v_j) = |\mathbb{G}_s(X)|$. That is to say, $\mathbb{G}_s$ satisfying oblivious traffic direction is a complete bipartite graph.*

Oblivious Traffic Size. Oblivious traffic direction alone is not sufficient to completely prevent privacy leakage in the distributed computing framework. If the adversary $\mathcal{A}$ has some background knowledge on the input data, it can still infer sensitive information by correlating actual traffic size with the background knowledge, such as the size distribution. It is necessary that the traffic size has to be identical for the entire process of job processing (i.e., by padding dummy data); otherwise, $\mathcal{A}$ is able to distinguish two jobs in polynomial time, as having been observed in [2] and [14]. We hence define the oblivious traffic size in $\mathbb{G}$ as follows.

Definition 3 (Oblivious Traffic Size). *In the directed graph $\mathbb{G}_{\mathrm{II}} = (V_{\mathrm{II}}, E_{\mathrm{II}})$ of job data-flow graphs, we denote the weight of an edge $\ell(v_i, v_j)$ as $w_{i,j}$, which represents the size of traffic from vertex v_i to v_j, where $v_i, v_j \in V_{\mathrm{II}}$. The graph $\mathbb{G}_{\mathrm{II}}$ satisfies oblivious traffic size, if for all $e \in E_{\mathrm{II}}$, we have $w_e = C$, where C is a constant.*

Note that the size of a transmitted data block may depend on system parameters. To keep all traffic size to be the same, we need to set the constant C sufficiently large to accommodate all possible data traffic.

4.3 Oblivious Distributed Computing - privacy

In this section, we propose Oblivious Distributed Computing - privacy (ODC-privacy), a definition of data privacy in distributed computing frameworks. Before giving the formal definition, we first show a security game for the oblivious traffic, which is performed by a malicious, adaptive adversary $\mathcal{A}$ and a challenger C. Given, $\mathcal{A}$ initially chooses two input datasets inp_0, inp_1 with equal size. The game proceeds as follows:

- *Setup:* In initialization, the challenger C fixes some framework and job parameters. It randomly generates a secret key $dk = \mathsf{KeyGen}(1^\lambda)$ for encrypting the traffic data.
- *Challenge:* The challenger C flips a coin and samples a random challenge bit $b \xleftarrow{\$} \{0, 1\}$. It encrypts the input set inp_b with the secret key dk and stores the ciphertext in the distributed computing framework. When the job starts to perform, the distributed computing framework ensures that traffic pattern in the entire job satisfies the requirements of oblivious traffic direction and oblivious traffic size by adding dummy messages and padding where necessary.
- *Query:* During performing the job, $\mathcal{A}$ is allowed to adaptively query C in the following three issues: (1) how many servers participate in the job; (2) what the phase of tasks server $\mathcal{P}$ performs; (3) what the size of traffic transmitted between two servers and between a user and a server. These three issues can be observed by an inner cloud adversary, which imply some passive attacks. Moreover, $\mathcal{A}$ can direct C to perform the following two operations: (1) arbitrarily drop intermediate tuples when necessary, which implies an active drop tuple attack; (2) exchange the output traffic of two tasks in the shuffle, which implies a data block misrouting attack;
- *Guess:* When the job terminates, $\mathcal{A}$ constructs job data-flow graphs $\mathbb{G}_b$ based on C's replies and outputs a bit b' to guess b. If $b' = b$, we say that the game outputs `true`, and otherwise it outputs `false`.

Now we propose the definition of ODC-PRIVACY:

DEFINITION 4 (ODC-PRIVACY). *The life-cycle of a job in distributed computing framework as defined in Section 4 is ODC-PRIVACY, if the following two properties are satisfied for any negligible function* $\mathsf{negl}()$*, and all large enough values of the security parameter* λ*:*

- ***Semantic security****: For any two messages* m_0, m_1 *of equal length, randomly choose one bit b from* $\{0, 1\}$*,* $c = \mathsf{Enc}\{k, m_b\}$*. The advantage of a PPT adversary* $\mathcal{A}$ *successfully guess which message is encrypted is less than* $\mathsf{negl}(\lambda)$*.*
- ***Oblivious traffic****: For all PPT adversaries, the advantage of winning the oblivious traffic game defined above,* $|\Pr[b' = b] - 1/2|$ *is less than* $\mathsf{negl}(\lambda)$*.*

5 OBLIDC: SGX-BASED OBLIVIOUS DISTRIBUTED COMPUTING FRAMEWORK

5.1 Design Details of ObliDC

In this section, we propose the oblivious distributed computing (ObliDC) framework based on Intel SGX (in Fig. 3). The framework consists of four subroutines $\pi_{JD}, \pi_{JI}, \pi_{JE}, \pi_{RR}$, and each of them is described as a two-party protocol. π_{JD} enables a user $\mathcal{U}$ to communicate with *each* server $\mathcal{P}$ to deploy the job code. π_{JI} allows the user $\mathcal{U}$ to upload input blocks to *each* task in the first phase. π_{JE} carries out a job execution, to allow one task (v_i) sending its outputs to another task (v_j) for further processing, denoted by an edge $\ell(v_i, v_j) \in \mathbb{G}_{\text{II}}$. π_{RR}, after the last phase tasks are finished, allows $\mathcal{U}$ to download the job outputs from the servers.

The ObliDC framework is designed to resist against passive and active attacks mentioned in Section 3.2. To counter ciphertext analysis, ObliDC uses either asymmetric or symmetric *IND-CPA* encryption schemes [19],[30]. Moreover, before data encryption, a hash value is appended to the plaintext to thwart malicious tampering by the adversary over the network. To against traffic analyses, ObliDC ensures its data shuffle satisfies both oblivious traffic direction and oblivious traffic size. That is, the framework guarantees that the outer degree of vertexes in $\mathbb{G}_s(X)$ is identical to $|\mathbb{G}_s(Y)|$, and the inner degree of vertexes in $\mathbb{G}_s(Y)$ is identical to $|\mathbb{G}_s(X)|$. Before job

```
$\mathrm{prog}_\pi[job, \mathcal{U}, \mathcal{P}_1, \ldots, \mathcal{P}_n]$
code$^+$: **On** input ("PKGen"): $(pk, sk) \leftarrow \mathsf{PKE.GEN}(1^\lambda)$; return pk
        **On** input ("Encrypt", $pt$, $k$): $ct = \mathsf{Enc}\{k, pt\}$; return $ct$
        **On** input ("Decrypt", $ct$, $k$): $pt = \mathsf{Dec}\{k, ct\}$; return $pt$
code$^-$: **On** input ("Compute", $d$, $y$): if $y \neq \perp$, return $y$; else $d = d - dummy$,
        $outp = f(d),\ outp = outp + dummy$; return $outp$

**Prot**$_\pi[job, \mathbb{G} = \{\mathbb{G}_{\text{I}}, \mathbb{G}_{\text{II}}\}, \mathcal{U}, \mathcal{P}_1, \ldots, \mathcal{P}_n, C]$
**Job Deployment** $\pi_{JD}$: for each $\ell(\mathcal{U}, \mathcal{P}_i) \in \mathbb{G}_{\text{I}},\ i \in \{1, 2, \ldots, n\}$:
$C$:   $dk = \mathsf{KeyGen}(1^\lambda)$;
       $C_{code} = \mathsf{Enc}\{dk, \text{code}^-\}$;
       sends the $C_{code}$ and code$^+$ to $\mathcal{P}$;
$\mathcal{P}$:   $eid = \mathtt{install}(sid, \text{code}^+)$;
       henceforth let $\mathcal{G}_{att}.\mathtt{resume}(\cdot) = \mathcal{G}_{att}.\mathtt{resume}(eid, \cdot)$;
       $(pk, \sigma) = \mathcal{G}_{att}.\mathtt{resume}(\text{"PKGen"})$;
       sends the $(pk, \sigma)$ to $C$;
$C$:   assert $\Sigma.\mathsf{Ver}_{mpk}\{\sigma\}$;
       $C_{key} = \mathsf{Enc}\{pk, dk\}$;
       sends the $C_{key}$ to $\mathcal{P}$;
$\mathcal{P}$:   $dk = \mathcal{G}_{att}.\mathtt{resume}(\text{"Decrypt"}, C_{key}, sk)$;
       code$^-$ $= \mathcal{G}_{att}.\mathtt{resume}(\text{"Decrypt"}, C_{code}, dk)$;
       sends "okay" to $C$;
**Job Initialization** $\pi_{JI}$: for each $\ell(\mathcal{U}, v) \in \mathbb{G}_{\text{II}},\ v \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$:
$C$:   If the size of $d$ is less than $C,\ d = d + dummy,\ d \in inp$;
      $C_d = \mathsf{Enc}\{dk, d\}$;
       sends $C_d$ to $v$;
$v$:   $d = \mathcal{G}_{att}.\mathtt{resume}(\text{"Decrypt"}, C_d, dk)$;
       $(\{outp_k\}_{k \in [\psi]}) = \mathcal{G}_{att}.\mathtt{resume}(\text{"Compute"}, d, \perp)$, where $\psi = d^-_{\mathbb{G}_{\text{II}}}(v)$;
**Job Execution** $\pi_{JE}$: for each $\ell(v_i, v_j) \in \mathbb{G}_{\text{II}},\ v_i, v_j \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$:
If $\ell(v_i, v_j) \in \mathbb{G}_s,\ \mathbb{G}_s \subseteq \mathbb{G}_{\text{II}}$, we must have $d^-_{\mathbb{G}_s}(v_i) = |\mathbb{G}_s(Y)|$ and $d^+_{\mathbb{G}_s}(v_j)$
$= |\mathbb{G}_s(X)|$.
$v_i$:   $C_d = \mathcal{G}_{att}.\mathtt{resume}(\text{"Encrypt"}, d, dk)$;
        sends the $C_d$ to $v_j$;
$v_j$:   await $\varphi$ input data $C_{d_i},\ i \in [\varphi]$, where $\varphi = d^+_{\mathbb{G}_{\text{II}}}(v_j)$;
        $d_i = \mathcal{G}_{att}.\mathtt{resume}(\text{"Decrypt"}, C_{d_i}, dk)$;
        $(\{outp_k\}_{k \in [\psi]}) = \mathcal{G}_{att}.\mathtt{resume}(\text{"Compute"}, \{d_i\}_{i \in [\varphi]}, \perp)$, where
        $\psi = d^-_{\mathbb{G}_{\text{II}}}(v_j)$;
**Results Return** $\pi_{RR}$: for each $\ell(v, \mathcal{U}) \in \mathbb{G}_{\text{II}},\ v \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$:
$v$:   $C_d = \mathcal{G}_{att}.\mathtt{resume}(\text{"Encrypt"}, d, dk)$;
       sends the $C_d$ to $C$;
$C$:   $d = \mathsf{Dec}\{dk, C_d\},\ d \in outp$;
```

Figure 3: Oblivious distributed computing (ObliDC) framework

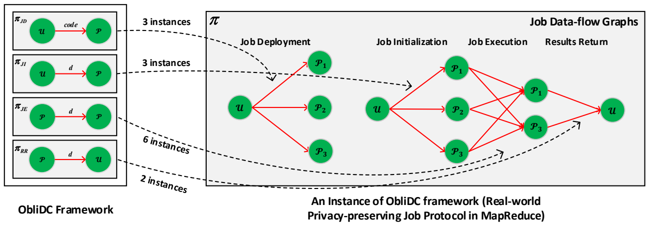


Figure 4: Modular construction for a real-world privacy-preserving MapReduce job protocol with three mappers and two reducers using ObliDC framework

execution, a constant C is chosen, and all data blocks in the network are padded to length C. If we encrypt these blocks in a semantically secure encryption scheme, the adversary can only distinguish two blocks from their ciphertexts with negligible probability.

A job code in ObliDC is divided into two parts: code^- and code^+. The former is the sensitive job code without being exposed, while the latter is the public shared code for each job including the public key generation PKE.GEN($\cdot$), encryption and decryption scheme $\text{Enc}\{k, \cdot\}$, $\text{Dec}\{k, \cdot\}$. Functions of the four subroutines are described as follows:

- *Job Deployment* π_{JD}: For each server $\mathcal{P}$ participating in a job, the user $\mathcal{U}$ generates a symmetric key dk for encrypting the sensitive data and code in authenticated encryption scheme (i.e., AES-GCM [31]). The code^- in ciphertext and the public code code^+ are then sent to $\mathcal{P}$. After receiving codes from $\mathcal{U}$, $\mathcal{P}$ first installs code^+ into the enclave and calls the PKGen($\cdot$) function to generate a pair of public and private keys (pk, sk) for securely delivering dk. The private key sk is kept in the isolated area, and the public key pk as well as the signature σ signed with $\mathcal{G}_{att}$'s private key msk are returned to $\mathcal{U}$ as a remote attestation (see Fig. 1 in detail). $\mathcal{U}$ verifies the correctness of σ using the $\mathcal{G}_{att}$'s public key mpk. If it succeed, $\mathcal{U}$ encrypts the symmetric key dk with pk and sends the C_{key} to $\mathcal{P}$'s enclave. $\mathcal{P}$ decrypts the ciphertext with sk first to get the symmetric key dk, and then decrypts C_{code} with dk to get the sensitive job code code^-.
- *Job Initialization* π_{JI}: For each task v in the first phase, the data block d as input is padded to the constant size C by $\mathcal{U}$ if necessary. d is then encrypted with dk and sent to $\mathcal{P}$'s enclave. v decrypts the data block by calling $\mathcal{G}_{att}$. It removes any dummy data and then executes a function f on this input. After finishing the task, the enclave pads each output block of f to the constant size C for all tasks in the next phase.
- *Job Execution* π_{JE}: Recall that an edge $\ell(v_i, v_j)$ in the graph $\mathbb{G}_\Pi$ represents the existence of a traffic transmission from v_i to v_j in which v_i encrypts one of v_j's input blocks d (the *outp* of v_i) in the enclave and delivers it to v_j, and v_j decrypts it and calls the sensitive code code^- as in job initialization. Note that a complicated job may consist of many phases of tasks; hence the process that intermediate results generated by tasks are sent to other tasks for further processing may repeat multiple times. From this point of view, a complicated execution process can be divided into multiple π_{JE} protocols.
- *Results Return* π_{RR}: For each task performed in the last phase, after the job is finished, the user $\mathcal{U}$ downloads the job outputs from the server $\mathcal{P}$ where task v is performed and decrypts it with the symmetric key dk.

Design of Backdoors in ObliDC. In ObliDC, we adopt two proof techniques in the UC framework. The first is *extraction*. In an attested execution secure processor, we plant a backdoor in the enclave program to allow $\mathcal{S}$ to extract the real inputs of corrupted servers. We assume that each server has a special "label". When $\mathcal{S}$ provides the correct label of a server, the enclave program will leak the real inputs of this server. However, this technique is not harmful to the security of honest servers, because no one learns honest parties' labels, including the honest parties themselves. The second is named *equivocation*, which plants a backdoor in $\mathcal{G}_{att}$ to enable the simulator $\mathcal{S}$ to sign on any data it needs during simulation. Recall that in the UC framework, the simulator $\mathcal{S}$ has no idea about the inputs of an honest server. It has to choose a canonical one and send to $\mathcal{G}_{att}$ on behalf of this server. While in the real world, the environment $\mathcal{Z}$ provides honest servers the real inputs. After the simulation terminates, the ideal functionality $\mathcal{F}$ evaluates on the canonical input provided by $\mathcal{S}$ and generates a false output, which can be different from the correct output *outp* in the real world. Moreover, $\mathcal{S}$ is unable to sign and modify $\mathcal{G}_{att}$'s signature because the private key msk is never revealed in public. This can be dangerous because it helps $\mathcal{Z}$ to distinguish the ideal world from the real world. To make the simulation successful, we enable the simulator $\mathcal{S}$ to program messages between a corrupt server and $\mathcal{G}_{att}$ by planting a trapdoor y inside the enclave program ("compute", d, y). If the trapdoor is $\perp$, it will compute on the input block d and generate a real output *outp*. Otherwise, $\mathcal{G}_{att}$ signs on y and returns y directly. This allows $\mathcal{S}$ to sign on any data it needs during simulation. Similar to the extraction technique, equivocation will not do harm to an honest server's security, because the trapdoor y is always $\perp$ in functions called by honest servers.

5.2 Modular constructions for concrete job protocols

The basic two-party protocols provided by ObliDC can be used to construct concrete privacy-preserving job protocols in distributed computing frameworks. For example, Fig. 4 shows a protocol instance of ObliDC. In this instance, three servers participate in performing a job. In job deployment, $\mathcal{U}$ communicates with all servers to distribute the job code and the symmetric key, which performs π_{JD} three times. That is, the real-world privacy-preserving job deployment protocol consists of three instances of π_{JD}. Similarly, in job initialization, $\mathcal{U}$ communicates with three tasks to be performed in the first phase, which means it has three instances of π_{JI}. During the shuffle process, because ObliDC framework is required to satisfy oblivious traffic direction, all tasks in the parent phase send data blocks to all tasks in the child phase. Hence, the real-world protocol has $3 \times 2 = 6$ instances of π_{JE}. For results return, it is obvious that the protocol contains two instances of π_{RR}.

Some security solutions for MapReduce (i.e., [2],[14]) also consider privacy leakage in network-level access pattern. In these previous works, all data blocks transmitted in the network are padded to the same size. For example, in [14], all mappers deliver intermediate results to all reducers to achieve oblivious traffic direction. In [2], similarly, all mappers send intermediate results to all *mixers*

```
$\mathcal{F}_{\pi}[sid, \mathbb{G} = \{\mathbb{G}_{\text{I}}, \mathbb{G}_{\text{II}}\}, \mathcal{U}, \mathcal{P}_1, \ldots, \mathcal{P}_n, C]$
Job Deployment: for each $\ell(\mathcal{U}, \mathcal{P}_i) \in \mathbb{G}_{\text{I}},\ i \in \{1, 2, \ldots, n\}$
Upon receiving code$^-$ from $\mathcal{U}$:
    notify $\mathcal{S}, \mathcal{A}$ of |code$^-$| and store code$^-$;
    send code$^-$ to $\mathcal{P}$;
    send a public delayed output "okay" to $\mathcal{U}$;
Job Initialization: for each $\ell(\mathcal{U}, v) \in \mathbb{G}_{\text{II}},\ v \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Upon receiving $d$ from $C$:
    notify $\mathcal{S}, \mathcal{A}$ of $C$;
    compute on $d$, $(\{outp_k\}_{k\in[\psi]}) = f(d)$, where $\psi = d^-_{\mathbb{G}_{\text{II}}}(v)$;
    send $\{outp_k\}_{k\in[\psi]}$ to $\psi$ subsequent tasks, respectively;
Job Execution: for each $\ell(v_i, v_j) \in \mathbb{G}_{\text{II}},\ v_i, v_j \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Upon receiving $d$ from $v_i$:
    await $\{d_i\}_{i\in[\varphi-1]}$ from $v_j$, where $d^+_{\mathbb{G}_{\text{II}}}(v_j) = \varphi$;
    if receive the number of input from $v_j$ less than $\varphi - 1$ then
        abort execution;
    notify $\mathcal{S}, \mathcal{A}$ of $C$;
    compute on $\varphi$ input blocks, $(\{outp_k\}_{k\in[\psi]}) = f(d, \{d_i\}_{i\in[\varphi-1]})$, where
$\psi = d^-_{\mathbb{G}_{\text{II}}}(v_j)$;
    send $\{outp_k\}_{k\in[\psi]}$ to $\psi$ subsequent tasks, respectively;
Results Return: for each $\ell(v, \mathcal{U}) \in \mathbb{G}_{\text{II}},\ v \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$
Upon receiving $d$ from $v$:
    notify $\mathcal{S}, \mathcal{A}$ of $C$;
    send a delayed output $d$ to $C$;
```

```
$\mathcal{F}_{JD}[sid, \mathcal{U}, \mathcal{P}]$
Upon receiving code$^-$ from $\mathcal{U}$:
    notify $\mathcal{S}, \mathcal{A}$ of |code$^-$|;
    send code$^-$ to $\mathcal{P}$;
    send a public delayed output "okay" to $\mathcal{U}$;
```

```
$\mathcal{F}_{JI}[sid, \mathcal{U}, v]$
Upon receiving $d$ from $C$:
    notify $\mathcal{S}, \mathcal{A}$ of $|d|$;
    compute on $d$, $(\{outp_k\}_{k\in[r]}) = f(d)$, where $r$ is the number of blocks sent by
$v$ in total in the real world;
    send $\{outp_k\}_{k\in[r]}$ to $r$ subsequent tasks, respectively;
```

```
$\mathcal{F}_{JE}[sid, v_i, v_j]$
Upon receiving $d$ from $v_i$, $v_i \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$:
    await $\{d_i\}_{i\in[t-1]}$ from $v_j$, where $t$ is the number of blocks received by $v_j$ in
total in the real world;
    notify $\mathcal{S}, \mathcal{A}$ of $|d|$;
    compute on $t$ input blocks, $(\{outp_k\}_{k\in[r]}) = f(d, \{d_i\}_{i\in[t-1]})$, where $r$ is the
number of blocks sent by $v_j$ in total in the real world;
    send $\{outp_k\}_{k\in[r]}$ to $r$ subsequent tasks, respectively;
```

```
$\mathcal{F}_{RR}[sid, \mathcal{U}, v]$
Upon receiving $d$ from $v$, $v \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$:
    notify $\mathcal{S}, \mathcal{A}$ of $|d|$;
    send a delayed output $d$ to $\mathcal{U}$;
```

Figure 5: Ideal functionalities of oblivious distributed computing and four subroutines in the ObliDC framework

first, and mixers then send permuted results to all reducers, which satisfies oblivious traffic direction as well. ObliDC is motivated by these previous efforts and can be regarded as a generic privacy-preserving distributed computing framework based on Intel SGX. Following this idea, more security solutions can be proposed in other frameworks, such as Microsoft Dryad [32] and Apache Tez [33].

5.3 Correctness of ObliDC

Theorem 1. *Assuming that both* code$^+$ *and* code$^-$ *are programmed correctly, and $\mathcal{G}_{att}$ properly performs these codes, then a protocol instance of the ObliDC framework, π generates a correct output under the passive attacks.*

Proof. Based on the observation that an adversary performing passive attacks who does not disrupt the operation of the ObliDC framework in any way, the proof is trivial, and we omitted here. □

6 FORMAL PROOF OF OBLIDC IN THE UC FRAMEWORK

In this section, we formally prove the security of a protocol instance π of the ObliDC framework. We first introduce the ideal functionality $\mathcal{F}_{\pi}$ for oblivious distributed computing and prove the security equivalence of $\mathcal{F}_{\pi}$ and ODC-privacy (Section 6.1). We then present ideal functionalities of four subroutines in ObliDC, denoted as $\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{JE}, \mathcal{F}_{RR}$, and prove each of them can be UC-realized by the corresponding subroutine, respectively (Section 6.2). Finally, we demonstrate that the protocol π is able to UC-realize $\mathcal{F}_{\pi}$ in $(\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{JE}, \mathcal{F}_{RR})$-hybrid model[2].

6.1 Security Equivalence of $\mathcal{F}_{\pi}$ and ODC-privacy

The ideal functionality $\mathcal{F}_{\pi}$ for oblivious distributed computation is shown in Fig. 5, which is partitioned into four phases. In job deployment, for each server $\mathcal{P}$, upon receiving code$^-$ from the user $\mathcal{U}$, $\mathcal{F}_{\pi}$ notifies the size of code to $\mathcal{S}$ and $\mathcal{A}$. After that, it stores the job code and returns "okay" to $\mathcal{U}$. In job initialization, for each task v in the first phase, after receiving one input block d from $\mathcal{U}$, $\mathcal{F}_{\pi}$ sends $\mathcal{S}$ and $\mathcal{A}$ the fake size of d — the constant C to satisfy oblivious traffic size. Output blocks $outp_1, \ldots, outp_{\psi}$ are later sent to other ψ units respectively, where ψ is equal to the outer degree

[2] We note that in some distributed computing frameworks, it is possible for some protocols to have tasks in only one phase. Hence, these protocols can only be composed by π_{JD}, π_{JI}, π_{RR} three subroutines, and the security will be proved in $(\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{RR})$-hybrid model. In this case, $\mathcal{F}_{\pi}$ will be re-designed as well. In this paper, for the generality, we choose more complicated protocols with tasks in more than one phase.

of v in $\mathbb{G}_\Pi$. In job execution, for each data traffic between two tasks v_i and v_j, upon receiving an input block from v_i, $\mathcal{F}_\pi$ needs to wait for other $\varphi - 1$ input blocks from v_j, where φ is the inner degree of v_j in $\mathbb{G}_\Pi$. If the number of input blocks $\mathcal{F}_\pi$ receiving from v_j is less than $\varphi - 1$, $\mathcal{F}_\pi$ will abort execution, which is directed by the oblivious traffic direction. Otherwise, $\mathcal{F}_\pi$ continues to compute on total φ input blocks and it notifies $\mathcal{A}$ and $\mathcal{S}$ the constant C. The same-sized output blocks will later be sent to tasks in the next phase, which number is identical to the outer degree of v_j. Finally, in the result return, upon receiving an input block d from the task v, $\mathcal{F}_\pi$ simply notifies $\mathcal{S}$ and $\mathcal{A}$ the constant C as the block size and delivers d to $\mathcal{U}$. We have

Theorem 2. *A job protocol π in the distributed computing framework UC-realizes $\mathcal{F}_\pi$ if and only if π is ODC-privacy.*

The proof of Theorem 2 is given in Appendix A.

6.2 Security of Subroutines in ObliDC

The four ideal functionalities $\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{JE}, \mathcal{F}_{RR}$ for the four subroutines of ObliDC are shown in Fig. 5. After receiving the job code code$^-$ from $\mathcal{U}$, $\mathcal{F}_{JD}$ directly delivers it to $\mathcal{P}$, notifies $\mathcal{S}$ and $\mathcal{A}$ the code size, and returns "okay" to $\mathcal{U}$. For $\mathcal{F}_{JI}$, upon receiving a input block d from the user, it tells $\mathcal{S}$ and $\mathcal{A}$ of the size of d and performs the task on d. The output blocks are later sent to r following tasks respectively. Note that a real-world job protocol always can be executed regardless of whether it is oblivious traffic direction. Hence, r here can be flexible and acceptable for a different number of input blocks in the real world without the restriction of ψ blocks in oblivious traffic direction. That is to say, r is possible to be any numbers less than ψ. For the ideal functionality of the job execution $\mathcal{F}_{JE}$, upon receiving an input block d from v_i, it has to wait other $t - 1$ blocks from v_j. Similarly, t is also dependent on the real-world execution and can be less than φ in $\mathcal{F}_\pi$. $\mathcal{F}_{JE}$ then works on these t input blocks and transfers r output blocks to r units respectively. For the last ideal functionality $\mathcal{F}_{RR}$, upon receiving an input block d, $\mathcal{F}_{RR}$ simply tells $\mathcal{S}$ and $\mathcal{A}$ the size of the block and delivers d to $\mathcal{U}$. In the following, we demonstrate that each subroutine is UC-secure in their corresponding ideal functionality. We have

Theorem 3. *Assuming all data transmission outside the TCB is encrypted in an authenticated encryption scheme, and the scheme is semantically secure, then the protocol $\pi_\alpha \in \{\pi_{JD}, \pi_{JI}, \pi_{JE}, \pi_{RR}\}$ UC-realizes $\mathcal{F}_\alpha$ in the presence of a malicious, adaptive adversary $\mathcal{A}$.*

The proof of Theorem 3 is given in Appendix B.

6.3 Security of Protocol Instance of ObliDC

In this section, we demonstrate the security of a real-world privacy-preserving job protocol π in distributed computing framework. First, we have

Theorem 4. *Let $\pi = (\pi_{JD}, \pi_{JI}, \pi_{JE}, \pi_{RR})$ be a protocol instance of ObliDC framework. Each subroutine ξ inside π UC-realizes its corresponding ideal functionality $\mathcal{F}_\xi$. We say that if π UC-realizes $\mathcal{F}_\pi$, the composed protocol $\pi^{\xi/\mathcal{F}_\xi}$ UC-realizes $\mathcal{F}_\pi$ as well, where $\pi^{\xi/\mathcal{F}_\xi}$ represents replacing an ideal functionality $\mathcal{F}_\xi$ called by $\mathcal{F}_\pi$ with a real protocol ξ UC-realizing $\mathcal{F}_\xi$.*

Table 2: Summary of applications used in evaluating ObliDC

Application	**LOC** (code$^-$)	**Size of enclave (mapper + reducer)**
Matrix Calculation	75	325KB + 331KB
K-means	117	375KB + 330KB
Monte Carlo Simulation	133	374KB + 330KB
WordCount	173	330KB + 331KB
RandomWriter	167	325KB + 331KB
Float-point Calculation	85	325KB + 330KB

Proof. This theorem can be demonstrated from universal composability directly. □

we now prove the distributed job protocol π UC-realizes $\mathcal{F}_\pi$ in $(\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{JE}, \mathcal{F}_{RR})$-hybrid model.

Theorem 5. *If all data transmission outside the TCB is encrypted in an authenticated encryption scheme, and the scheme is semantically secure, then the protocol π as an instance of ObliDC framework UC-realizes $\mathcal{F}_\pi$ in $(\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{JE}, \mathcal{F}_{RR})$-hybrid model with the presence of a malicious, adaptive adversary $\mathcal{A}$. That is,*

$$\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \overset{c}{\equiv} \mathsf{IDEAL}_{\mathcal{F}_\pi,\mathcal{S},\mathcal{Z}}$$

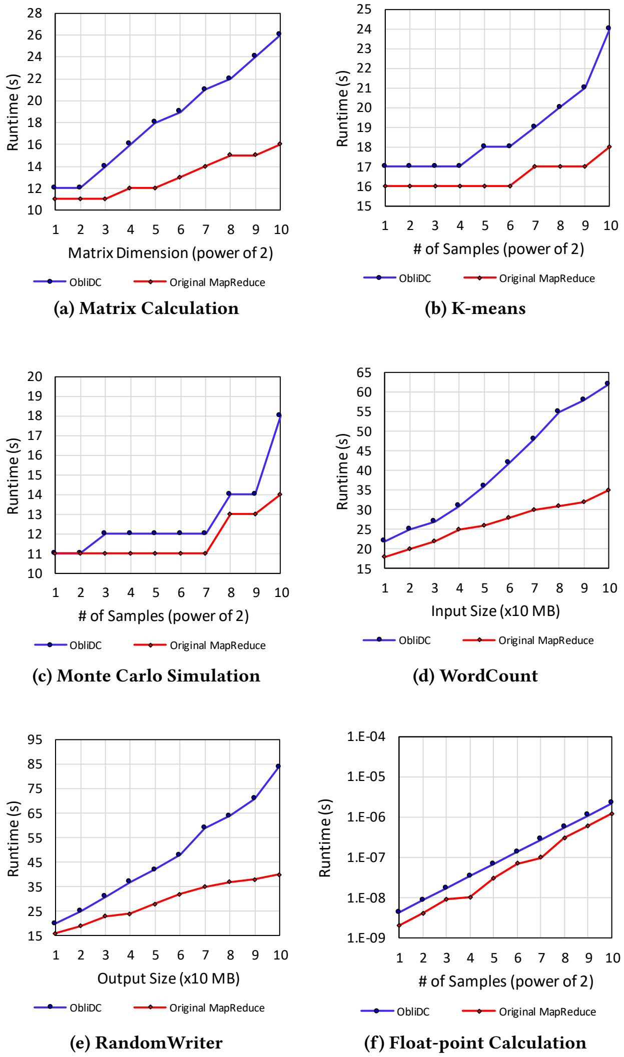
The proof of Theorem 5 is given in Appendix C.

7 EXPERIMENTS AND EVALUATION

To assess the efficiency of ObliDC, we realize the framework in MapReduce and compare its performance with the original MapReduce in several applications. These applications are listed in Table 2.

ObliDC in MapReduce and Applications. To be compatible with the SGX code in `C` language, we measure the performance of ObliDC on `Hadoop Streaming`. We choose six real-world applications and realize them using version 2.1 Intel SGX SDK [26]. For applications realized in Intel SGX, we define sensitive job code code$^-$ in the `.edl` file, and all data blocks in the network are encrypted using AES-GCM (we use AES-NI instructions to implement the encryption scheme). Before showing our results, we first briefly introduce each application:

(1) *Matrix Calculation*: Randomly choose two large matrices with the same dimension and calculate their product. Each element in both matrices ranges from 100 to 1000.
(2) *K-means*: Randomly choose a series of points (x, y) as samples in the Cartesian coordinate system. Given the number of clusters and iterations, make clusters for these points. We use Euclidean distance in the calculation.
(3) *Monte Carlo Simulation*: Randomly choose a series of points (x, y) as samples in a 2×2 square in the Cartesian coordinate system. Count the number of points resident in the unit circle and statistically estimate the value of Pi.
(4) *WordCount*: Count the number of occurrences for each word in a text set.
(5) *RandomWriter*: Randomly generate a sequence of strings with a predefined length.
(6) *Float-point Calculation*: For each sample, randomly generate two decimal numbers with four places and perform their addition and multiplication once respectively.



(a) Matrix Calculation

(b) K-means

(c) Monte Carlo Simulation

(d) WordCount

(e) RandomWriter

(f) Float-point Calculation

Figure 6: Comparison of execution time of applications running in ObliDC and in the original MapReduce

Experimental Setup. We perform our experiments on a server equipped with 3.00GHz Intel Xeon E3-1220 v6 CPU, 16GB RAM and, 100GB disk. We create a cluster with ten virtual machines as worker nodes, and each of them is under Ubuntu 16.04 LTS operation system. The Hadoop version we use is 2.7.2.

Comparison in Different Applications. As shown in Fig. 6, in these six applications, jobs performed in ObliDC have stable running time, with overheads between 33%-110% over the original MapReduce. The overhead is mainly due to three factors: (1) Encryption and decryption in the enclave. After reading task input into the secure region, mappers and reducers have to decrypt first. Before task results leaving from the enclave, they are encrypted again to keep confidentiality. Moreover, all data in the enclave are also encrypted by Memory Encryption Engine (MEE). Before executing CPU instructions and accessing the encrypted memory in an enclave, the processor has to decrypt these data first; (2) Context switching and enclave transitions. During entering and exiting an enclave (performing ECALL and OCALL functions), CPU has to load contents from memory into the cache, and it will lead to a performance impact. Similarly, the system interrupt is another reason leading to a performance reduction. If it happens, OS has to store all program states at the breakpoint and recover these states before performing the program again; (3) Oblivious shuffle for a large number of intermediate results. To hide the traffic access pattern of the shuffle, ObliDC produces extra dummy messages and the size of intermediate results are larger than in original MapReduce, both of which will bring more overheads because the real mapper outputs have to take a long time before been transferred to the reducer.

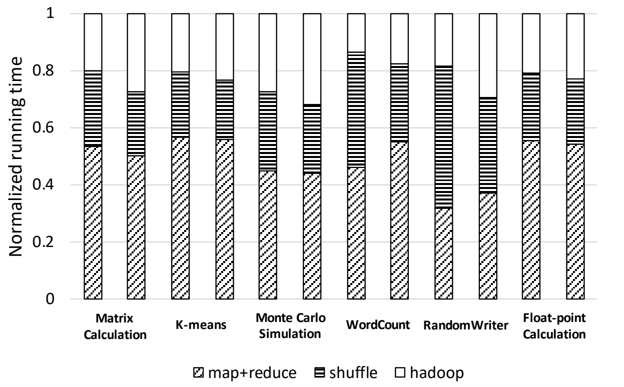


Figure 7: Normalized running time in phases of job for applications realized in ObliDC and original MapReduce (for each application, the left bar shows the normalized time in ObliDC and the right bar is in original MapReduce)

From the entire job running time, it also can be seen that the overheads in some IO-intensive applications (i.e., WordCount and RandomWriter) are more apparent than other CPU-intensive applications. In WordCount and RandomWriter, massive intermediate results have to be encrypted and padded to the same size. Before performing reduce tasks, all dummy data has to be removed again, which takes a long time (e.g., about 27s in WordCount and 44s in RandomWriter). While in CPU-intensive applications, the overheads can be much lower because these applications normally generate fewer intermediate results (e.g., mappers in Monte Carlo Simulation only output several numbers, which bring 4s overheads merely).

Overheads in Job Phases. As depicted in Fig. 7, we observe that the oblivious shuffle (including oblivious traffic direction and oblivious traffic size) using in ObliDC is significant, especially in IO-intensive jobs. The overhead is up to 40.4% in WordCount and 50.1% in RandomWriter. The cost primarily due to the privacy-preserving shuffle of a large amount of data. We have to choose a significant large constant number as the traffic size to accommodate all network traffic. While in some CPU-intensive jobs, such as K-means, Monte Carlo Simulation and Float-point Calculation, the overheads in shuffle are much lower because of their few map task outputs, which only between 23.7%-27.6% over the entire running time. Moreover, in these CPU-intensive applications, the costs of mappers and

Table 3: Comparison of distributed computing security solutions

Security Solutions	Against network-level access pattern	Genetic framework	Formally Proved in ObliDC
Haven [18]	×	✓	×
VC3 [1]	×	×	✓
M^2R [2]	✓	×	✓
Ohrimenko et al. [14]	✓	×	✓
Opaque [17]	✓	×	×
ObliDC	✓	✓	✓

reducers in enclave are relatively larger than that of the oblivious shuffle. The second observation is that compared with applications realized in original MapReduce, shuffle processes of ones realized in ObliDC take more time, which ranges from 1.03× to 1.51× as the insecure shuffle. As having analyzed before, oblivious shuffle used in ObliDC generates a large amount of dummy data to hide the network traffic access pattern. Both traffic direction and traffic size are oblivious under the conditions of traffic padding. Thus, it takes more time to shuffle intermediate results than original MapReduce.

8 RELATED WORK

Haven [18] is the first to propose combining Intel SGX with the untrusted cloud. Before executing the program, Haven has to load the whole Windows 8 OS library into an enclave, which makes a large TCB. Moreover, the system cannot guarantee data integrity during computation in the cloud. VC3 [1] is another SGX-based secure system designed on Hadoop MapReduce. Instead of loading the whole OS library into the enclave as Haven, VC3 only keeps sensitive code and data in the TCB. Verifiers in VC3 guarantee all outputs from the tasks (or users) are not modified in the network, which protects data integrity. However, although all messages are encrypted in the network, VC3 is unable to protect against privacy leakage from network-level access patterns [13]. M^2R is a privacy-enhanced system based on VC3, which modifies the shuffle process in cascaded mix network [34]. Ohrimenko et al.[14] also find VC3 insecure during shuffling and improves its security by Melbourne shuffle [35]. However, both M^2R and VC3 only focus on the MapReduce framework and the shuffle process instead of the genetic framework and the life-cycle of the job in ObliDC. Opaque [17] is a security system based on Spark SQL. It protects data privacy from both network and memory level access pattern. The system rewrites some operators in Spark SQL to make access pattern oblivious during enclave computing. All these existing efforts have not provided rigorous formal security proofs for their systems. A comparison of previous distributed computing security solutions and ObliDC is given in Table 3.

Some other works are devoted to formalizing trusted processor. Considering trusted processors like Intel SGX are hard to prevent from side-channel attacks, Tramèr et al. [36] propose a new model for trusted processors, named *transparent enclave*, in which all secrets and states of the application in the enclave are revealed to the adversary during execution. They also show that some security protocols such as commitment schemes and zero-knowledge proofs can be realized with transplant enclave. Similarly, Pass et al. [3] propose a formal abstraction for genetic trusted processor $\mathcal{G}_{att}$. Instead of revealing inner states to adversary like transplant enclave, $\mathcal{G}_{att}$ keeps all sensitive information inside the enclave such as secret key msk. During formal proofs in GUC framework, $\mathcal{G}_{att}$ works as a global trusted setup functionality, which makes it sharable by many protocols. While in transplant enclave, the setup is assumed to be "local" and some enclave information like master keys cannot be reused in protocols. ObliDC allows enclave to create attestations for more jobs with a single private key, hence we choose $\mathcal{G}_{att}$ in our proofs. Unlike our work, both of these previous work study some common secure computation protocols instead of focusing on a specific application framework, such as distributed computation. We take $\mathcal{G}_{att}$ as a primitive during formal proofs, and further propose a model of distributed computing framework and a general privacy-preserving framework. Subramanyan et al. [37] propose a formal verification method for trusted hardware platforms. Instead of proving the security of protocols as mentioned above, they focus on program execution inside the enclave. They show how to prove the security of remote attestation and how a trusted hardware platform satisfies integrity, confidentiality, and secure measurement. The differences are that they focus on trusted hardware platforms instead of distributed computing frameworks, and they only care about inner-enclave security and some security mechanisms of the trusted processor, while the security of job protocols takes center stage in our work.

9 CONCLUSION AND FUTURE WORK

In this paper, we proposed an oblivious distributed computing framework, named ObliDC, which allows for modular construction of job protocols in distributed computing based on Intel SGX with provable security in the UC framework. We first modeled the life-cycle of a job as job data-flow graphs by the data transmission between a user and a task as well as between tasks. In this graph model, to prevent traffic analysis, we proposed two preconditions: oblivious traffic direction and oblivious traffic size, as countermeasures. We further formally defined the notion of data privacy in distributed computation, named ODC-PRIVACY. Operations of the life-cycle of a job are specified by four subroutines in the ObliDC framework, which are designed to satisfy ODC-PRIVACY. Inspired by $\mathcal{G}_{att}$ [3], a formal abstraction for the trusted processor, we formally proved the security of the four subroutines and that of the job protocol π composed of the four subroutines in the presence of adaptive adversaries. As applications of ObliDC, we provided formal security proofs of VC3 [1] and M^2R [2]. In ObliDC, we mainly considered the scenarios in distributed computation. While in other fields like anonymous communications, some systems are proposed with the help of Intel SGX as well, such as SGX-Tor [38]. It will be interesting to adapt our ObliDC framework to other fields and formally prove their security.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant No.61672062, 61232005, U1804263, 61702105 and AXA Research Fund in Singapore.

REFERENCES

[1] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *IEEE S&P*, pages 38–54. IEEE, 2015.

[2] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2r: Enabling stronger privacy in mapreduce computation. In *USENIX Security*, pages 447–462, 2015.

[3] Rafael Pass, Elaine Shi, and Florian Tramer. Formal abstractions for attested execution secure processors. In *EuroCrypt*, pages 260–289. Springer, 2017.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *USENIX HotCloud*, 10(10-10):95, 2010.

[6] Apache storm, 2018. https://storm.apache.org/index.html.

[7] U.s. federal cloud computing market forecast 2015–2020, 2015. http://www.marketresearchmedia.com/?p=145.

[8] Cisco global cloud statistic: Forcasting and methods, 2015–2020, 2015. https://www.cisco.com/c/dam/m/zh_cn/solutions/service-provider/sp_gciwhitepaper_whitepaper_cn.pdf.

[9] The cloud computing and distributed systems (clouds) laboratory, 2014. http://www.cloudbus.org/.

[10] Mobile & cloud computing laboratory (mobile & cloud lab), 2014. http://mc.cs.ut.ee/.

[11] Advanced encryption standard (aes), 2008. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[12] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[13] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[14] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *CCS*, pages 1570–1581. ACM, 2015.

[15] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[16] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC*, pages 420–443. Springer, 2010.

[17] Wenting Zheng, Ankur Dave, Jethro Beekman, Raluca Ada Popa, Joseph Gonzalez, and Ion Stoica. Opaque: A data analytics platform with strong security. In *USENIX NSDI*, 2017.

[18] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX OSDI*, pages 267–283, 2015.

[19] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In *CRYPTO*, pages 26–45. Springer, 1998.

[20] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948. SIAM, 2010.

[21] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud. *Technical Report*, 2014.

[22] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE, 2001.

[23] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[24] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13. ACM, 2013.

[25] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[26] Intel software guard extensions sdk for linux os (version 2.1), 2018. https://download.01.org/intel-sgx/linux-2.1/docs/Intel_SGX_Developer_Reference_Linux_2.1_Open_Source.pdf.

[27] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In *TCC*, pages 61–85. Springer, 2007.

[28] Yehida Lindell. Secure multiparty computation for privacy preserving data mining. *The Journal of Privacy and Confidentiality*, 1(1):59–98, 2005.

[29] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, pages 640–656. IEEE, 2015.

[30] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *AsiaCrypt*, pages 531–545. Springer, 2000.

[31] David McGrew and John Viega. The galois/counter mode of operation (gcm). *Submission to NIST Modes of Operation Process*, 20, 2004.

[32] Microsoft dryad, 2018. https://www.microsoft.com/en-us/research/project/dryad/.

[33] Apache tez, 2018. http://tez.apache.org/.

[34] Marek Klonowski and Miroslaw Kutylowski. Provable anonymity for networks of mixes. In *International Workshop on Information Hiding*, pages 26–38. Springer, 2005.

[35] Olga Ohrimenko, Michael T Goodrich, Roberto Tamassia, and Eli Upfal. The melbourne shuffle: Improving oblivious storage in the cloud. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 556–567. Springer, 2014.

[36] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *IEEE EuroS&P*, pages 19–34. IEEE, 2017.

[37] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A formal foundation for secure remote execution of enclaves. In *CCS*, pages 2435–2450. ACM, 2017.

[38] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. Enhancing security and privacy of tor's ecosystem by using trusted execution environments. In *USENIX NSDI*, pages 145–161, 2017.

A PROOF OF THEOREM 2

Theorem 2 Revisited. *A job protocol π in the distributed computing framework UC-realizes $\mathcal{F}_\pi$ if and only if π is ODC-privacy.*

Proof. For the "only if" direction, assuming π is not ODC-privacy, we show that it is unable to UC-realize $\mathcal{F}_\pi$. This can be done by constructing an environment machine $\mathcal{Z}$ and an adversary $\mathcal{A}$ in the real world. For any *PPT* simulator $\mathcal{S}$, $\mathcal{Z}$ can distinguish whether it is dealing with π and $\mathcal{A}$ in the real world or $\mathcal{S}$ and $\mathcal{F}_\pi$ in the ideal world. We discuss in the following two cases:

1) *π is not semantically secure.* Given the ciphertext of a message m in the network, let $|m|$ denote the message length. In the ideal world, upon a receiving a message, $\mathcal{F}_\pi$ sends the length of the message to the simulator $\mathcal{S}$, which is the only information obtained by $\mathcal{S}$ about the message. Hence, $|m|$ is a part of $\mathcal{S}$'s view and is written to $\mathcal{S}$'s output tape. Formally, let $\mathsf{IDEAL}_{\mathcal{S}}^{\mathcal{F}_\pi}(\lambda, inp)$ and $\mathsf{REAL}_{\mathcal{A}}^{\pi}(\lambda, inp)$ denote the view of $\mathcal{S}$ and $\mathcal{A}$ in the ideal and real world, respectively, and we further have $|m| \in \mathsf{IDEAL}_{\mathcal{S}}^{\mathcal{F}_\pi}(\lambda, inp)$. However, in the real world, if a message in π is not encrypted by a semantically secure encryption scheme, then a *PPT* adversary $\mathcal{A}$ can exist who can extract more information about m, and we denoted by m'. Then both $|m|$ and m' are written to the output tape of $\mathcal{A}$, i.e., $|m|, m' \in \mathsf{REAL}_{\mathcal{A}}^{\pi}(\lambda, inp)$. Hence, $\mathcal{Z}$ is able to distinguish the ideal world and the real world from the views of $\mathcal{S}$ and $\mathcal{A}$ with a non-negligible probability.

2) *π is not oblivious traffic.* In this case, the adversary $\mathcal{A}$ is able to distinguish whether $\mathbb{G}_b$ is $\mathbb{G}_0$ or $\mathbb{G}_1$ in the polynomial time as follows:

• *Difference in Graph Structure.* For a given job and fixed system parameters, assuming that sizes of two inputs inp_0 and inp_1 are equal but π does not satisfy oblivious traffic direction, we then have $\exists \mathbb{G}_s \subseteq \mathbb{G}_b$, where $\exists v_i \in \mathbb{G}_s(X), \exists v_j \in \mathbb{G}_s(Y), \ell(v_i, v_j) \notin E$. In other words, $\exists v_i \in \mathbb{G}_s(X), d_{\mathbb{G}_s}^{-}(v_i) < |\mathbb{G}_s(Y)|$ and $\exists v_j \in \mathbb{G}_s(Y), d_{\mathbb{G}_s}^{+}(v_j) < |\mathbb{G}_s(X)|$. For a data flow $\ell(v_i, v_j)$ in the shuffle process of π, if the task v_i makes output blocks less than $|\mathbb{G}_s(Y)|$, v_j will receive input blocks less than $|\mathbb{G}_s(X)|$. In the ideal world, $\mathcal{S}$ simulates the operation of the real world only if it provides $\varphi = |\mathbb{G}_s(X)|$ input blocks to $\mathcal{F}_\pi$. Otherwise, the simulation won't be continue. However, in the real world, π always can execute regardless of the number of input blocks, which makes a difference to the ideal world and helps $\mathcal{Z}$ distinguish these two worlds. Moreover, traffic analysis in the real world can help $\mathcal{A}$ extract more information than $\mathcal{S}$. Thus, in this issue, $\mathcal{Z}$ can tell whether it is dealing with the ideal world or the real world.

- *Difference in Edge Weights.* It is apparent that $\mathcal{A}$ will distinguish $\mathbb{G}_0$ and $\mathbb{G}_1$ based on weights of edges if π is not oblivious traffic size. In the ideal world, the ideal functionality $\mathcal{F}_\pi$ always pads the size of data blocks to the constant C before transmitting. While in the real world, if π is not oblivious traffic size, the size of some block can be less than C. Therefore, we have $\exists|m| \in \mathsf{IDEAL}^{\mathcal{F}_\pi}_{\mathcal{S}}(\lambda, inp), \exists|m'| \in \mathsf{REAL}^{\pi}_{\mathcal{A}}(\lambda, inp), |m| \neq |m'|$. If $\mathcal{A}$ has some background knowledge, it can infer more information than $\mathcal{S}$ in the ideal world. This difference in output tapes of $\mathcal{A}$ and $\mathcal{S}$ allows $\mathcal{Z}$ to distinguish the ideal world and the real world.

For the "if" direction, assume that π does not UC-realize $\mathcal{F}_\pi$, then it is not ODC-PRIVACY. In this case, a *PPT* $\mathcal{Z}$ exists, which can distinguish transcripts of $\mathcal{S}$ and $\mathcal{A}$ generated in the ideal world and the real world, respectively. During the simulation of π, $\mathcal{S}$ only receives block size from $\mathcal{F}_\pi$ in the ideal world, while $\mathcal{A}$ can extract extra information in the following three ways. Firstly, after intercepting an encrypted data block, if the encryption scheme is not semantically secure, $\mathcal{A}$ can infer more information other than the block size. Secondly, in the ideal world, $\mathcal{F}_\pi$ can continue its work only if it has received φ input blocks. If it fails, π will not be UC-secure any more, which makes a chance for $\mathcal{A}$ to perform traffic analyze to learn more privacy. Finally, if $\mathcal{A}$ can distinguish $\mathbb{G}_0$ and $\mathbb{G}_1$ by observing the size of data traffic over the network, then more information is written to the output tape of $\mathcal{A}$ if it has some background knowledge about the input data, and π does not satisfy oblivious traffic size required in oblivious traffic. □

B PROOF OF THEOREM 3

Theorem 3 Revisited. *Assuming all data transmission outside the TCB is encrypted in an authenticated encryption scheme, and the scheme is semantically secure, then the protocol $\pi_\alpha \in \{\pi_{JD}, \pi_{JI}, \pi_{JE}, \pi_{RR}\}$ UC-realizes $\mathcal{F}_\alpha$ in the presence of a malicious, adaptive adversary $\mathcal{A}$.*

Proof. Here, we demonstrate the subroutine π_{JE} can UC-realize $\mathcal{F}_{JE}$ as an example. Proofs of other subroutines are similar, and we omit here. We assume that any communication between $\mathcal{Z}$ and $\mathcal{A}$ or between $\mathcal{A}$ and $\mathcal{G}_{att}$ is simply forwarded to $\mathcal{S}$. According to the communication process between v_i and v_j, we discuss in the following four cases and construct $\mathcal{S}$ for each of them:

- *Case 1: v_i is honest before sending C_d and v_j is honest when receives C_d.* In this case, both of v_i and v_j are honest during π_{JE} execution and $\mathcal{S}$ will perform operations on behalf of them. First, $\mathcal{S}$ generates a canonical data block d' for v_i in the ideal world and sends it to $\mathcal{F}_{JE}$ as an input. After $\mathcal{S}$ calling ("Encrypt", d', dk) in the real world, v_j is honest as well. $\mathcal{S}$ then generates other $\varphi - 1$ canonical input blocks for v_j and delivers all of them to $\mathcal{F}_{JE}$. After all of the blocks having prepared, $\mathcal{F}_{JE}$ works on φ blocks and generates a result *outp*. The indistinguishability proof of this case is trivial as all communication between v_i and v_j is assumed to occur over secure channels. The eavesdropper only extracts the length of a message transmitted in the channel. In this case, the simulated operation of $\mathcal{S}$ is identically distributed as the real execution in case 1.
- *Case 2: v_i is honest before sending C_d to v_j and v_j is corrupt when receives C_d.* Similar to the case 1 except that $\mathcal{Z}$ provides another $\varphi - 1$ input blocks for v_j instead of $\mathcal{S}$. Moreover, in case 2, $\mathcal{A}$ calls the function ("Decrypt", C_d, dk) to decrypt the input block for v_j in enclave instead of $\mathcal{S}$. If $\mathcal{A}$ makes a call ("Compute", $\{d_i\}_{i\in[\varphi]}, \perp$), $\mathcal{S}$ will extract all real inputs by the extract trapdoor in the enclave program and delivers them to $\mathcal{F}_{JE}$. After the ideal functionality $\mathcal{F}_{JE}$ generating *outp*, $\mathcal{S}$ replaces the message ("Compute", $\{d_i\}_{i\in[\varphi]}, \perp$) from $\mathcal{A}$ to $\mathcal{G}_{att}$ with ("Compute", $\{d_i\}_{i\in[\varphi]}, outp$) by equivocation backdoor, and forwards the response ($outp, \sigma$) to $\mathcal{A}$. Because of the restriction of oblivious traffic direction and oblivious traffic size, the adversary $\mathcal{A}$ receives the identical number of messages as the simulator $\mathcal{S}$ in the ideal world. Additionally, all data blocks are encrypted in semantically secure encryption scheme, which leads to both $\mathcal{A}$ and $\mathcal{S}$ only know the constant block length C. Hence, the views of $\mathcal{A}$ and $\mathcal{S}$ are indistinguishable for taking passive attacks. For the active attacks, although $\mathcal{A}$ is able to exchange block traffics from two tasks during shuffle, this will not help it extract more information because π_{JE} satisfies both two preconditions perfectly. While in the ideal world, $\mathcal{S}$ forwards all extracted input blocks to $\mathcal{F}_{JE}$, which outputs the same result *outp* as the real world. If $\mathcal{A}$ arbitrarily drops one data block in shuffle, $\mathcal{S}$ will simply drops the corresponding block in the ideal world, which makes the same result as well. Based on these analyses, no matter $\mathcal{A}$ takes passive or active attacks, $\mathcal{S}$ will perfectly simulate its operations, and views of them are indistinguishable in polynomial time.
- For another two cases: *Case 3: v_i is corrupt before sending C_d to v_j and v_j is honest when receives C_d* as well as *Case 4: v_i is corrupt before sending C_d to v_j and v_j is corrupt when receives C_d* are similar to the case 2 above. The security of π_{JE} in both cases can be reduced to the semantic security of encryption scheme and oblivious traffic as well. We omit them for the brevity. □

C PROOF OF THEOREM 5

Theorem 5 Revisited. *If all data transmission outside the TCB is encrypted in an authenticated encryption scheme, and the scheme is semantically secure, then the protocol π as an instance of ObliDC framework UC-realizes $\mathcal{F}_\pi$ in ($\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{JE}, \mathcal{F}_{RR}$)-hybrid model with the presence of a malicious, adaptive adversary $\mathcal{A}$. That is,*

$$\mathsf{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \overset{c}{\equiv} \mathsf{IDEAL}_{\mathcal{F}_\pi,\mathcal{S},\mathcal{Z}}$$

Proof. We construct a simulator $\mathcal{S}$ for $\mathcal{F}_\pi$ in the ideal world. It works as follows:

Simulating the communication with $\mathcal{Z}$. All input data received from $\mathcal{Z}$ will be wrote to the input tape of $\mathcal{A}$, and the output of $\mathcal{A}$ is copied to the output tape of $\mathcal{S}$ as well. Moreover, $\mathcal{S}$ is able to achieve all messages between $\mathcal{A}$ and $\mathcal{G}_{att}$.

Simulating four subroutines in π. See cases of simulator $\mathcal{S}$ constructions in Section 6.2.

Simulating corruptions of servers. This can be proved by discussing different cases of simulator $\mathcal{S}$ constructions in corrupting different servers during job protocol. We can show the views of $\mathcal{A}$ and $\mathcal{S}$ are indistinguishable for each case. Due to the space limitation, we omit the proof here.

We have declared that the protocol π performed by a malicious, adaptive adversary $\mathcal{A}$ and n servers $\mathcal{P}_1, \ldots, \mathcal{P}_n$ in the real world generate identical-distribution transcripts with $\mathcal{S}$ and $\mathcal{F}_\pi$ in the ideal world. In other words, no *PPT* $\mathcal{Z}$ can distinguish whether it contacts with the ideal world or the real world. $\mathcal{S}$ operates by running a simulated copy of $\mathcal{A}$. For proving the computational indistinguishability, we put all cases above together and analyze in the following four aspects: input, output, intermediate messages of servers in the view of $\mathcal{S}$ and $\mathcal{A}$ as well as corrupting method in two worlds.

(1) *Input*: In all cases above, if a server is corrupt, during $\mathcal{G}_{att}$ computing, $\mathcal{S}$ is able to extract the real input by the trapdoor inside the enclave program and sends to $\mathcal{F}_\pi$. Due to the fact that all messages are transmitted over the secure channel in the ideal world and the encryption scheme in the real world is semantically secure, it is obvious that inputs of corrupt servers are the same under both of these two worlds.

(2) *Output*: For all subroutines in π, if outputs of some corrupt servers are generated by sensitive code computation ("Compute", $\cdot, \perp$) (e.g., v in job initialization or v_j in job execution), the simulator $\mathcal{S}$ will replace the message by ("Compute", $\cdot$, *outp*), where *outp* is generated by $\mathcal{F}_\pi$ in the ideal world. This equivocation will generate identical outputs as the ideal world.

(3) *Intermediate messages*: A copy of $\mathcal{A}$ is maintained inside the $\mathcal{S}$. Hence, all messages sent by $\mathcal{A}$ during protocol perform are copied to the output tape of $\mathcal{S}$, and messages received by $\mathcal{A}$ are copied to the input tape of $\mathcal{S}$ as well. Therefore, both distributions of messages received by $\mathcal{A}$ and sent from $\mathcal{A}$ in the real world are identical to the ideal world.

(4) *Parties Corruption*: All communication between $\mathcal{Z}$ and $\mathcal{A}$ are forwarded to $\mathcal{S}$. If $\mathcal{Z}$ requires $\mathcal{A}$ to corrupt a server in the real world, $\mathcal{S}$ will "corrupt" the corresponding server in the ideal world and communicate with $\mathcal{F}_\pi$ on behalf of the server. For all corruptions in the real world, $\mathcal{S}$ can also realize them in the ideal world.

Based on analyses above, we can conclude that the protocol π UC-realizes $\mathcal{F}_\pi$ in $(\mathcal{F}_{JD}, \mathcal{F}_{JI}, \mathcal{F}_{JE}, \mathcal{F}_{RR})$-hybrid model. □

D FORMAL PROOFS FOR VC3 AND M^2R

Some previous works such as VC3 [1] and M^2R [2] are security solutions based on distributed computing framework. However, both of them only adopts heuristic security proof and semi-formal reasoning [3], lacking formal proofs of security. In this section, based on the formal model and ideal functionalities before, we give formal proofs for these previous works.

Theorem 6. *We denote the life-cycle of a job in VC3 as a protocol $\pi^{VC3} = (\pi_{JD}^{VC3}, \pi_{JI}^{VC3}, \pi_{JE}^{VC3}, \pi_{RR}^{VC3})$. We say that if π^{VC3} UC-realizes the ideal functionality $\mathcal{F}_\pi$ if and only if VC3 has a secure shuffling process.*

Proof. We denote the job data-flow graphs formed by π^{VC3} as $\mathbb{G}^{VC3} = \{\mathbb{G}_{\mathrm{I}}^{VC3}, \mathbb{G}_{\mathrm{II}}^{VC3}\}$. By analyzing job performing in VC3, we find two differences between π^{VC3} and the privacy-preserving protocol constructed by ObliDC framework in this paper: (1) For the subroutine π_{JE}^{VC3} in π^{VC3}, it does not consider the secure shuffle. The adversary $\mathcal{A}$ can extract more information about the sensitive data by traffic analysis. While other three subroutines are identical to corresponding ones in π. (2) The size of messages in VC3 are not identical at all. They lack some padding data, so the adversary $\mathcal{A}$ may infer some privacy if it has some background knowledge. Hence, $\mathbb{G}_{\mathrm{II}}^{VC3}$ satisfies neither oblivious traffic direction nor oblivious traffic size. Formally speaking, $\exists \mathbb{G}_s^{VC3} \subseteq \mathbb{G}_{\mathrm{II}}^{VC3}, \mathbb{G}_s^{VC3}$ is not a complete bipartite graph. Moreover, in $\mathbb{G}_{\mathrm{II}}^{VC3} = (V_{\mathrm{II}}^{VC3}, E_{\mathrm{II}}^{VC3})$, $\exists e_1, e_2 \in E_{\mathrm{II}}^{VC3}, w_{e_1} \neq w_{e_2}$, if we denote the size of transmitted message by the weight of edge in the graph.

Straightforwardly, we research on the transmitted messages in the shuffle process. During job execution in VC3, the system only protects the confidentiality of data by encrypting message contents and putting tasks execution in the isolated area, while it ignores network-level access pattern. In the shuffle process, one mapper may not send data blocks to all reducers, which makes $\mathbb{G}_s^{VC3}$ not a complete bipartite graph. Furthermore, $\exists v_i \in \mathbb{G}_s(X), d^-_{\mathbb{G}_s^{VC3}}(v_i) < |\mathbb{G}_s(Y)|$ and $\exists v_j \in \mathbb{G}_s(Y), d^+_{\mathbb{G}_s^{VC3}}(v_j) < |\mathbb{G}_s(X)|$. One mapper generates output blocks less than $|\mathbb{G}_s(Y)|$ which results in one reducer has less than $|\mathbb{G}_s(X)|$ input blocks. During $\mathcal{S}$ simulates operations with $\mathcal{F}_\pi$ in the ideal world, $\mathcal{S}$ will only choose less than $|\mathbb{G}_s(X)|$ canonical data blocks for v_j, if v_j is honest; If v_j is corrupt, $\mathcal{S}$ can only extract less than $|\mathbb{G}_s(X)|$ data blocks from $\mathcal{G}_{att}$ by the trapdoor within as well. In both of these cases, $\mathcal{F}_\pi$ will not perform and abort execution because it does not have enough input blocks. The *outp* can be calculated only if $\mathcal{F}_\pi$ has $\varphi = |\mathbb{G}_s(X)|$ data blocks. For (2), in the ideal world, all messages are padded to the same size C, while they can be any size less than C in the real world. Therefore, it is obvious that $\mathcal{Z}$ is able to distinguish the transcripts from the real world and ideal world by the size of received messages in them. According to the analysis above, $\mathcal{Z}$ has the capability to tell whether it contacts with the real world or the ideal world with non-negligible probability. □

Theorem 7. *We denote the life-cycle of a job in M^2R as a protocol $\pi^{M^2R} = (\pi_{JD}^{M^2R}, \pi_{JI}^{M^2R}, \pi_{JE}^{M^2R}, \pi_{RR}^{M^2R})$. We say that π^{M^2R} is able to UC-realize the ideal functionality $\mathcal{F}_\pi$.*

Proof. We denote the job data-flow graph formed by π^{M^2R} as $\mathbb{G}^{M^2R} = \{\mathbb{G}_{\mathrm{I}}^{M^2R}, \mathbb{G}_{\mathrm{II}}^{M^2R}\}$. In this security solution, all mappers' output blocks pass through several mixers, which randomly permutes all received messages in the enclave. Combined with the formal model before, we conclude that $\mathbb{G}_{\mathrm{II}}^{M^2R}$ is identical to the graph formed by ObliDC framework. For each instance of $\pi_{JE}^{M^2R}$ executed in the ideal world, $\mathcal{S}$ is able to achieve enough data blocks from the real world and send to $\mathcal{F}_\pi$. It will not abort execution as in VC3. In further, all messages transmitted in the network are encrypted using a semantically secure encryption scheme. The plaintext is padded with dummy data to the same size. Hence, the only information $\mathcal{A}$ extracts from the ciphertext is the constant size C. If $\mathcal{A}$ performs some active attacks, such as drop tuple attack and data block misrouting attack, they will not help the adversary obtain more information because both oblivious traffic direction and oblivious traffic size are satisfied in $\mathbb{G}_{\mathrm{II}}^{M^2R}$. According to analyses above, the protocol π^{M^2R} can be perfectly simulated in the ideal world, and none *PPT* $\mathcal{Z}$ can distinguish the ideal world and the real world. Hence, we say that π^{M^2R} is secure in the UC framework. □