ThriftyMPC: Reducing the Cost of Large-Scale MPC in the Cloud

David Inyangson[1], Sahbaaz Ansari[2], Tushar M. Jois[3], Rosario Gennaro[3], Gamze Gürsoy[4,5], Gabriel Kaptchuk[6], Moti Yung[4,7] and Diogo Barradas[8]

[1] Johns Hopkins University, Baltimore, United States
[2] University of Luxembourg, Luxembourg City, Luxembourg
[3] City College of New York, New York, United States
[4] Columbia University, New York, United States
[5] University of Cambridge, Cambridge, United Kingdom
[6] University of Maryland College Park, College Park, United States
[7] Google, New York, United States
[8] University of Waterloo, Ontario, Canada

Abstract. Cloud computing has become the standard for large-scale computation, offering elastic scalability and on-demand resources that exceed typical on-premise capabilities. However, many large-scale computations over *sensitive data* – such as genome-wide association studies (GWAS) – face significant barriers to cloud adoption due to privacy concerns and regulatory constraints. While cryptographic primitives like multi-party computation can alleviate these concerns through provable privacy guaranties, their substantial communication and computational overhead can make cloud deployment cost-prohibitive. To address both privacy and cost constraints, we present ThriftyMPC. ThriftyMPC is a framework that leverages spot instances (ephemeral cloud compute at reduced rates) to enable cost-effective, privacy-preserving computation at scale by combining secure multi-party computation with preemption-tolerant execution. We introduce a formal model for multi-party execution under ephemeral compute conditions, demonstrate how ThriftyMPC handles spot instance preemptions while maintaining cryptographic security guaranties, and provide a formal discussion of these guaranties. Our evaluations on realistic GWAS-inspired workloads over the Google Cloud Platform demonstrate robust execution despite spot instance churn, and show significant cost reduction compared to the state-of-the-art multi-party computation framework (MP-SPDZ) run traditionally using on-demand instances. We show that leveraging multi-party computation on spot instances makes privacy-preserving computation economically viable, enabling organizations to harness the cloud for sensitive workloads previously confined to isolated, on-premise deployments.

Keywords: multi-party computation · cloud computing · genomics · spot instances

1 Introduction

Data science is most useful when it is *collaborative*. For instance, in a genome-wide association study (GWAS), data scientists attempt to identify genetic variants associated

E-mail: dinyang1@jhu.edu (David Inyangson), 02327614530@uni.lu (Sahbaaz Ansari), tjois@ccny.cuny.edu (Tushar M. Jois), rosario@ccny.cuny.edu (Rosario Gennaro), gg584@cam.ac.uk (Gamze Gürsoy), kaptchuk@umd.edu (Gabriel Kaptchuk), motiyung@gmail.com (Moti Yung), diogo.barradas@uwaterloo.ca (Diogo Barradas)

with biological traits. A large, multi-cohort GWAS allows for analysis across populations to identify markers for rare diseases [MR19]; modern GWAS interrogate millions of genetic variants simultaneously, the vast majority of which have small effect sizes and many of which occur at low frequencies in the population [CWR+20]. As a result, statistical power scales primarily with sample size, making cross-institutional aggregation not merely beneficial but essential. Genomic data, however, is highly sensitive and tightly regulated, and institutional, legal, and ethical constraints often prevent direct data sharing. Indeed, there has been a renewed push for privacy *across* data science: limiting access to raw data to only the party that owns it, mostly due to the ethical issues surrounding massive datasets [SD19, GG16, Pas17].

In this context, *secure multiparty computation* (MPC) [Yao86, GMW87, BGW88, CCD88, GRR98] is a promising solution to private collaborative data science. These cryptographic protocols allow parties to compute a joint functionality without sharing their private inputs. MPC has the right model of computation – federating trust between mutually distrustful parties computing a shared function on sensitive data – and practical frameworks are plentiful [MNPS04, HHNZ19, ACC+18, Kel20, BDST22]. There is also industry interest in deploying MPC, as seen in AWS's Clean Rooms service [Her23].

At the same time, data science is incentivized to minimize cost. Most collaborative data science happens in the cloud, due to its clear benefits for availability, global access, and low up-front cost [LN18]. Given the size of modern datasets [LRK+18, MR19], however, data scientists are pursuing ways to reduce their costs on the cloud [KH20, KDNGS+20, BHS+17], and in the extreme case repatriating cloud data back to local servers [Zgo23]. This, in turn, *disincentivizes* data scientists from adopting MPC, given that it adds non-trival runtime overhead [DIK10]. In an illuminating example, a study of AWS Clean Rooms showed that it was as expensive as running AWS's *most expensive instance* [KSC23]. While fundamental cryptography for MPC will undoubtedly improve, the cost of MPC is impacting its deployment for data science.

Given this state of affairs, to improve the adoption of MPC for cloud data science, we must develop ways to reduce its cost in the cloud. While protocol optimizations are useful, considering the architecture and hosting environment is also important, as argued by Katz and Koo [KK07]. In the cloud setting, an interesting direction emerges from the systems community, which has studied *spot instances* as a way to reduce cloud costs [SLG+15, WUG+17, WMLP20, GMT+22, SGS+15, XSDW16]. These instances represent the cloud provider's excess compute capacity, and can be priced much lower than standard "on-demand" instances by up to 90% [GCP26a, AWS25]. Spot instances stand to provide significant cost savings, especially for long-running workloads like MPC. But, importantly, these instances can be *preempted* at any time and forced to shut down with little warning. Preemption requires rethinking the MPC execution model and adapting it to spot compute.

Contributions. In this paper, we initiate the study of techniques to reduce the cost of large-scale MPC through the use of spot instances. We make the following contributions:

- We translate dynamic MPC, a primarily theoretical primitive, into practice by building cloud-native *preemption tolerance strategies* that can be deployed over spot instances.
- We design preemption-tolerance strategies that allow a variety of honest-majority t-out-of-n underlying MPC primitives to maintain security under arbitrary preemption.
- Along the way, we develop a new definition, *Checkpoint-Dynamic MPC*, which improves the flexibility of dynamism through the checkpointing and transfer of state.
- Using our preemption-tolerance strategies, we build ThriftyMPC, the *first cloud-native end-to-end practical MPC framework for spot instances*
- We evaluate ThriftyMPC on realistic GWAS workloads over the Google Cloud Platform (GCP) showing that despite overhead induced by its preemption tolerance

mechanisms, ThriftyMPC can achieve *cost savings of 27–70%* over modern optimized MPC frameworks.

2 Background and Related Work

We begin by providing background on topics relevant to our work, as well as related work in the area.

Secure multi-party computation. In MPC, parties collectively compute a functionality $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$, where X_i denotes the input space of the i^{th} party (P_i) and Y_i denotes its corresponding output space. The functionality f is typically represented as a circuit, which may consist of addition and multiplication gates in the case of arithmetic circuits, or Boolean gates in the case of Boolean circuits. While there are many templates for realizing MPC, the most well-studied approaches in the applications literature are from linear secret sharing schemes. In Shamir's (t, n)-threshold secret-sharing scheme [Sha79a], a secret $s \in \mathbb{F}$ (for some finite field $\mathbb{F}$) is shared among n parties, yielding a sharing $[s]$ by sampling a degree t polynomial

$$p(x) = s + a_1 x + a_2 x^2 + \cdots + a_t x^t \in \mathbb{F}[x],$$

and setting P_i's share to be $[s]_i = p(i)$. The scheme has the property that any $t+1$ parties can reconstruct s, while any coalition of at most t parties learns no information about s. Informally, security in MPC requires that no coalition of corrupted parties learns anything beyond what can be inferred from their own inputs and outputs of the computation.

In an MPC protocol based on Shamir secret sharing, each input $x_i \in X_i$ is first secret-shared among the parties using the (t, n)-threshold scheme described above. Linear functions of shares (i.e., addition in arithmetic circuits) can be evaluated locally using the linearity of the secret sharing scheme as $[s] + [s'] = [s + s']$. Multiplication, however, requires interaction: the product of two degree-t sharing polynomials has degree at most $2t$. Thus, after local multiplication of shares, the parties can engage in a *degree-reduction* (or "resharing") protocol [GRR98] to obtain a fresh sharing of the product under a degree-t polynomial. Parties reconstruct outputs $y_1, \ldots, y_m$ by sending their shares of $[y_i]$ to P_i.

Practical MPC. There has been a continuous push for practical MPC frameworks for general purpose computation [MNPS04, HHNZ19]. MPC can be written in languages such as C [HFKV12, ZSB13, ZE15, MGC+16], C++ [WMK16], Java [LWN+15, Ale], Python [Sch18], and JavaScript [Mul]. Each framework provides different guarantees for inputs based on the underlying theoretical construction; e.g., MPyC [Sch18] provides semi-honest security via Shamir's secret sharing [Sha79b], while FRESCO [Ale] provides dishonest majority malicious security via SPDZ [KSS13]. There has also been significant work optimizing MPC, both at the protocol [KSS13, ACC+18, DSZ15] and systems [BLW08, Kel20, BDST22] levels. These efforts have allowed MPC to be applied to the real-world, from measuring the wage gap [LJA+18] to Danish sugar beet auctions [BCD+09] to privacy-preserving machine learning [MZ17, ZPGS19].

Spot instances. Spot instances are spare cloud compute resources that service providers like Amazon Web Services (AWS) and Google Cloud Platform (GCP) [AWS25, GCP26a] offer clients at significant savings, which can reach up to 90% . In this setting, a user places a *bid* for an instance, and once its *spot price* is lower, the instance starts. The success of a particular bid request is dependent on the capacity pool – the set of unused cloud compute for a particular compute resource – and market conditions. Prices fluctuate based on the demand for resources within a capacity pool [AWS25].

In contrast to on-demand instances, which persist until terminated, spot instances are ephemeral. When the spot price goes higher than the bid price (i.e., due to increased demand), the spot instance is *reclaimed* (or *preempted*), and will be forcefully shut down.

Cloud providers typically give advance warnings (1) when market conditions signal spot instances are *likely* to be revoked and (2) when the service provider will *definitively* revoke the spot instance. The time between the notice and the actual revocation is is *revocation window*. When a spot instance is revoked, all running jobs are lost unless action is taken to save their state [AWS25, GCP26a].

While spot reclamation makes long-running computation challenging, through the creation of preemption tolerance strategies, systems researchers have been able to build reliable services on top of spot instances for batch jobs [MSJ14, SGS+15], in-memory storage [XSDW16, WUG+17], machine learning inference [WMLP20, GMT+22], and even an entire derivative cloud platform [SLG+15]. Generally, though, the security & privacy community has yet to use spot instances in applications, save for a recent work that uses them for censorship resistance [KKP+24b].

3 Intuition

In designing ThriftyMPC, we try to resolve the competing incentives surrounding the use of MPC for data science by investigating *spot instances* and *dynamic MPC*.

Reducing costs with optimized primitives. A natural direction for reducing the cost of cloud-assisted MPC is to leverage optimized MPC frameworks, which can reduce the runtime of an execution, resulting in less hours billed on the cloud. For example, MP-SPDZ [Kel20] achieves these gains through a combination of techniques including preprocessing, batched communication, and vectorized operations. These system-level optimizations, however, are ultimately constrained by the underlying protocol design. Motivated by this limitation, recent work has revisited efficient 3PC and 4PC protocols, proposing schemes such as Astra [CCPS19], Swift [KPPS21], Fantastic Four [DEK21], and Tetrad [KPRS22]. These protocols take advantage of the properties of replicated secret sharing to enable more efficient core operations for computations with a smaller number of parties. Efficiency here, however, comes at the cost of flexibility since these protocols are not directly applicable to an arbitrary t-out-of-N setting.

Reducing costs with cloud architecture. Protocol optimizations are important, but we take a complementary approach. When embedding an MPC protocol into a hosting architecture, considering the implementation requirements *within* that architecture (and not leaving the actual environment as a cost-free abstraction) contributes to realizing and optimizing the real implementation cost.

Katz and Koo first advocated this point of view in 2007 [KK07]. Their observation concerned the cost of implementing MPC in a point-to-point network where global agreement is often needed to be solved. Similarly, we argue that the overall performance and cost of an MPC implementation cannot depend merely on the protocol as implemented in an abstract sense; instead, we must also consider the cloud, its basic primitive offerings, and their associated costs.

We embrace this view, designing general *cloud-native* mechanisms that can be layered on top of existing t-out-of-N MPC protocols for cost savings. As discussed in Section 2, *spot instances* are an attractive way to save on computing costs, with reductions up to 90%. This much lower price comes with the caveat that the instance can be *preempted* and forced to shut down at any time. In traditional MPC, an interrupted computation means that the entire computation must be started from the beginning, and any resources that were used up to that point are essentially wasted. Traditional MPC therefore cannot be directly run on spot instances.

To build real systems on top of spot instances, we take inspiration from the systems community, which has seen success in doing so. The systems literature shows that we must support *preemption tolerance*: gracefully handling spot instance reclamation and allowing computation to continue. Preemption tolerance methods can be classified as follows. First,

migration offloads a job (i.e., the entire execution state of the node's memory) to a new instance [SLG+15, WUG+17, WMLP20, GMT+22]. Second, *checkpointing* periodically records each job's state to storage, allowing for another node to continue from the last state of a revoked node. Checkpointing requires additional storage overhead than migration, but is not bound by the revocation window, so it can handle recovering from jobs with a larger memory footprint [SGS+15]. [1]

Dynamism in MPC. Given the success the systems community has seen in using migration and checkpointing, it follows that they are promising ways to achieve MPC on spot instances. But, traditional MPC is limited by the assumption that the participating parties are *static*. Each party that starts the protocol must continue to completion, which is not reasonable especially if we wish to consider spot instances.

Notions of *dynamism* are a way to overcome this limitation. An early example is *proactive MPC*, in which computational parties regularly refresh their state to mitigate the ability of a strong but slow moving adversary to exceed the privacy threshold [OY91, BELO14]. More recent work has built upon this to consider models of *churn*, i.e., nodes explicitly entering and leaving the protocol during protocol execution [GPS19, BBG+21, DEP21, BJMS20, CGG+21, GHK+21], while maintaining security. Two common approaches to securely handling churn (and addressing proactive security) are the Layered MPC and YOSO models.

Layered MPC [CGG+21, DKLZN24, DDG+23] models computation as a directed acyclic graph, where a committee of nodes is responsible for executing a portion of the computation. In this model, each committee only needs to communicate with the committees immediately preceding and following it. The adversary is stronger in this setting, as it is allowed to corrupt up to a threshold t of servers in every committee. This model assumes a deterministic upper bound on adversarial corruption per committee, which models a targeted attack against compute nodes during a computation – a realistic scenario in the cloud-assisted setting.

The YOSO [GHK+21, EMP26, AHKP22, CDGK22] line of work typically considers environments with a large pool of compute nodes. An adversary can compromise a significant portion of this pool, but only a small subset of nodes is randomly selected at any given time to perform part of the computation. Additionally, this subset of nodes is only required to speak once, and can afterwards immediately leave the computation. Because the adversary does not know which nodes are active at a given moment, it is limited to probabilistic corruptions during any round. While this model has certain advantages, it relies on assumptions like target-anonymous-channels and is not directly applicable to the deterministic t-out-of-N corruption settings that we aim to address. YOSO is designed for decentralized environments where adversarial control can spread to a large fraction of nodes; in our cloud setting, an adversary is unlikely to corrupt a significant portion of a cloud provider's spot instance infrastructure, across multiple zones and regions, at any given time. For these reasons, we tailor our design to the Layered MPC model.

Challenges. Our goal is to design a protocol within the cloud itself employing the cheapest computational resources the cloud offers. This involves mixing (1) the protocol primitives, (2) the cloud's operations, and primarily developing (3) new protocol components that mediate between the abstract MPC protocol and the cloud environment.

With this in mind, to reduce cost in the cloud, we can use spot instances, and to use spot instances, we can apply preemption-tolerance strategies via Layered MPC approaches. Our analysis and implementation indeed require all three components above, as well as the interdisciplinarity of mixing basic protocols at an implementation layer with the setting of cloud operations and their properties and costs.

[1] Another approach, *replication* [SGS+15, XSDW16], simply runs the same job across several cloud regions to reduce the impact of preemption. But it is probabilistic, relatively unreliable, and requires more assumptions on the cloud provider, and so we do not consider it.

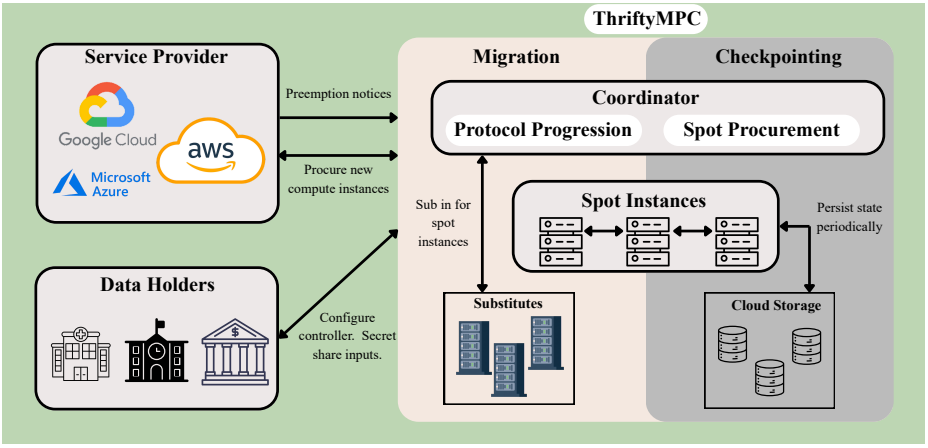


Figure 1: An overview of ThriftyMPC.

To build this system, we must addresses the following challenges:

The computation model of dynamic MPC does not fully align with that of spot instances. Layered MPC is expressed in terms of computation rounds: parties enter at the start of a round, a computation occurs, and parties either continue in the next round or exit by handing-off to others. The presence of spot instances, on the other hand, does not neatly fit into a round-based model: we no longer have the assumption of atomic rounds because preemption can happen at any point, even in the middle of a computation round. Moreover, there is no guarantee on how long it takes to procure a node that can replace one that has been revoked. If a replacement cannot quickly be procured, all progress may be lost. Thus, the Layered MPC view of churn is different than the reality of spot architecture. These differing views necessitate additional infrastructure and strategies to realize preemption tolerance.

Realizing preemption tolerance introduces security implications for MPC. Layered MPC literature exists primarily in the theoretical setting, and thus relies on assumptions that are abstracted away from practical deployment. Translating these assumptions into concrete system mechanisms directly impacts the security of the resulting protocol. For a specific example, consider the simple scenario where a spot instance is preempted and therefore needs to be replaced. In Fluid MPC [CGG+21], for instance, the strategy of how and when to select a replacement is modeled as a sampling function, but the details of realizing this in a practical deployment are unspecified.

On the other hand, the systems works on spot instances typically propose utilizing a *coordinator node* to advance protocol progression by procuring new spot instances from the cloud provider and facilitating handoffs. But, the coordinator in these prior systems can see data in the clear, a non-starter for MPC. Even if data is hidden from a coordinator, the mere presence of the coordinator impacts security – the typical model of MPC is that the only parties executing the protocol are computing nodes. While there is research on non-dynamic, helper-assisted MPC [MK22, KKP+24a, IPP+22], and dynamic homomorphic encryption [MCPT24], there is nothing matching the model of Layered MPC with a coordinator node. Other security implications of preemption-tolerance design include how to safely incorporate checkpointing strategies, and how to safely migrate state within cloud provider revocation windows.

Checkpointing has not been studied as a form of dynamism in MPC. In Layered MPC works such as Fluid MPC [CGG+21], churn is handled at the end of a round, wherein a computing party hands off state to another party, i.e., fitting the model of migration. But, if we want to apply checkpointing to MPC, this is flipped: now, churn occurs when nodes

rewind to the beginning of the round in response to a reclamation. Checkpointing, as mentioned above, is more flexible than migration, but has not been studied in the dynamic MPC literature. We must particularly understand the security impact of resuming from a previous step of the computation, especially in the case that other nodes have completed that step.

Dynamic MPC has not yet seen a transition to practice. Most dynamic MPC protocols are considered only theoretically and implementations are limited to microbenchmarks (e.g., [CGG+21]). No framework contains a dynamic MPC implementation, which means it has not been deployed for practical analysis. As such, we do not have a concrete understanding of the overheads inherent to dynamic MPC (e.g., handoffs between parties) over traditional MPC. This issue is related to the others, as a lack of implementations precludes deployment and therefore understanding deployment considerations.

Our approach. These challenges reflect an important lesson from the systems community: we must *adapt* the workload to match the architecture of the cloud to maximize benefits and minimize costs; simply "lifting and shifting" the workload is insufficient [TUS11].

With the above in mind, in the rest of this paper, we adapt Layered MPC to spot instances (Section 4) by designing a framework for secure preemption tolerance, describe security under both migration and checkpointing preemption tolerance strategies (Section 5), and highlight our framework's end-to-end deployability and cost savings on realistic workloads (Section 6). A high-level overview of our system design is displayed in Figure 1.

Threat model. In designing our system, we wish to meet the threat model most useful to cloud data science, an application that stands the most to gain from cheaper access to MPC. We derive our threat model and adversarial assumptions from GWAS, a representative example of collaborative science on sensitive data. The most common adversarial model adopted in the GWAS literature assumes that participating institutions and the cloud provider are semi-honest [CWB18a, MMD19] and therefore we use it here. This assumption reflects the realities of large-scale genomic collaboration: participating institutions are typically bound by legal agreements, ethical review processes, and regulatory frameworks that strongly discourage active protocol deviation. Moreover, these parties have aligned incentives – namely, to obtain correct and reproducible scientific results – and little to gain from sabotaging a joint computation whose outputs they themselves rely on. As such, given the sensitive nature of GWAS, information disclosure is the primary risk.

The sheer scale of GWAS further motivates the semi-honest assumption [MMD19]. Modern GWAS involve millions of variants and large cohorts, making fully maliciously secure MPC prohibitively expensive in many realistic deployments. In this setting, the semi-honest model represents a pragmatic tradeoff that enables analyses at the required scale while still providing strong confidentiality guarantees against passive inference. Providing malicious security while retaining the cost-efficiency we desire is an interesting direction for future work. At the same time, we briefly discuss in Section 5 a path towards achieving malicious security.

The primary adversarial actor is a network-based one that corrupts cloud instances. The adversary could be acting as a party, an insider at the cloud provider, or another user of the cloud (e.g., through colocation attacks on the cloud [RTSS09, VZRS15, XWW15, ZMFT24]). We consider an adversary who can corrupt instances over the course of computation, but instances that are renewed (e.g., due to spot instance churn) are no longer corrupted, similar to the notion of proactive security [OY91, BELO14]. We assume that the adversary can corrupt no more than t parties at any given time during the computation (and therefore cannot reconstruct the secrets from shares). This assumption stems from the general difficulty of an adversary corrupting *multiple* nodes in a computation undetected, and is the classic threshold assumption in MPC.[2]

[2]To reduce the risk that a semi-honest cloud provider could breach the threshold assumption (e.g., by

We consider large-scale spot market manipulation attacks with the intent of raising spot prices, increasing churn, or exceeding corruption thresholds to be out of scope. In practice, such risks can be mitigated by diversifying across instance types and regions, and are further constrained by the infrequent price updates in GCP. We provide additional details on our security model in Section 5.

4 System Design

ThriftyMPC is a framework that leverages custom preemption tolerance mechanisms to adapt Layered MPC approaches to spot instance architecture. The computation in ThriftyMPC follows the Layered MPC client-server model where a set of clients each hold their own private inputs and offload these inputs to a set of servers. Like traditional MPC protocols, these servers collectively compute a functionality $f : X_1 \times \cdots \times X_m \rightarrow Y_1 \times \cdots \times Y_m$, where X_i denotes the input of the i^{th} server and Y_i denotes its corresponding output. This functionality f is represented as an arithmetic *circuit* composed of addition and multiplication gates.

Computation proceeds in rounds where in each round a set of servers participate. We refer to this set of servers as a *committee*. Each server in a committee is *associated with a client*, which is a party submitting its dataset to be a part of the MPC. Because committee members are spot instances and can be preempted, successful protocol progression requires addressing three aspects of computation: (1) dynamic committee membership, (2) circuit design and execution, and (3) preemption tolerance. The rest of this section addresses these topics.

4.1 Dynamic Committees

In traditional MPC, committees are static – once the computation starts executing, all servers must complete the circuit to the end. But, spot instances are subject to preemption, so using them as servers for MPC requires the support of *dynamic committees*, where committee changes may happen during any round of the computation. Taking inspiration from the dynamic MPC literature, we divide each round into two phases: a compute and a handoff. In the compute phase, servers perform computation over their local states. In the handoff phase, servers transfer their local state to the servers that make up the next committee, if necessary due to a committee change. Unlike prior work of Layered MPC, however, in ThriftyMPC a committee change may occur at any part of the round (due to spot preemption), even during the compute phase. We will address recovering from this in Section 7.2.

Now, because the committees are dynamic in ThriftyMPC, we must be able to keep track of which spot instances are a part of the committee, so we can gracefully recover from spot preemption. Providing this type of synchronization is left unspecified in the Layered MPC literature. To solve this problem, we leverage a *coordinator* who "manages" spot instances: taking action to provide maximal availability of these servers in the presence of churn.

The coordinator interacts with all running spot instances and maintains a global view of protocol progress, but is not trusted with access to any party's secret shares or private state. In particular, the coordinator cannot compute on, store, or access any sensitive data. Its role is limited to orchestration, such as signaling computation rounds and provisioning replacement instances. We assume that the coordinator is consistently online, i.e., is not a spot instance.

introspecting all of the parties' VMs and reconstructing shares), we can spread out the parties between cloud providers such that no more than t are on any one cloud.

To enforce this isolation, we leverage cloud-native identities (e.g., Service Accounts in GCP). Clients are associated with a unique service account, and all storage and cloud permissions are bound to that identity. Because spot instances are associated with clients, spot instances run within the context of a specific Service Account. When a spot instance is preempted, the coordinator provisions a replacement instance and assigns it the same service account, allowing it to recover the corresponding party's state without exposing it to the coordinator or other instances. Crucially, the coordinator is not granted permission to impersonate or access these service accounts (only to provision instances with them), and thus cannot access any protected resources.

This isolation is useful; in practice, it allows for the coordinator (who cannot access any sensitive information) to be managed jointly by the data-holding clients. In theory, it also allows us to model the coordinator as a synchronization primitive, useful for proving the security of our framework. We also note that all communication is end-to-end authenticated and encrypted. GCP natively encrypts all traffic between VMs and additionally provides support for mutual TLS. Thrifty MPC embraces cloud features such as these to achieve a more realistic deployment of MPC on commercial clouds.

Prior to computation, clients, e.g., data-holding institutions, can provision the coordinator with the instance types, regions, and zones from which spot instance compute should be procured. The coordinator provisions spot instances on behalf of users according to these set parameters. We implement this on GCP using their REST API (Section 6). Due to spot market fluctuations, it is possible that a similar spot instance cannot be procured in the same cloud region according to user-given parameters. In these cases, the coordinator can procure replacements of different types or within other regions until market conditions improve.

4.2 Circuit Design and Execution

We represent each MPC functionality f as an arithmetic circuit consisting of addition and multiplication gates. While existing MPC compilers and languages are well established [HHNZ19, Kel20, BDST22, BMIMT+23, BHK+23], they do not provide the abstractions needed for preemption tolerance. To address this, we design a custom circuit description language for ThriftyMPC, implemented in Rust, that offers the additional language support required for this setting, namely a `handoff()` primitive. An example is shown in Listing 1. Supporting existing circuit formats and representations like FUSE [BHK+23] would involve extending their compilation pipelines to interpret the intermediate representation as a layered dependency graph and introduce `handoff()` operations at appropriate layer boundaries.

Naïvely assigning the entire circuit of a large-scale computation like GWAS to the first committee of the protocol can result in the computation failing to progress once a preemption occurs – a wasted computation in the ThriftyMPC setting. To enable finer-grained progress, we employ *layered circuits*, in which the outputs of layer i serve as the inputs to layer $i + 1$. In an arithmetic circuit, additions and scalar operations require no communication, whereas multiplications require a degree reduction [GRR98] and communication between computing servers. So, in practice, each layer i typically ends in a series of multiplications that are necessary as inputs for layer $i + 1$. This approach to layerization is the one taken by Fluid MPC [CGG+21], which also notes that the degree reduction step [GRR98] is an appropriate place to handle changes to the committee.

As a first step, we could simply use a layerized circuit, and compute one layer per round ℓ. But, in ThriftyMPC, we run our computations on spot instances. If a preemption notice is issued during a round, there is no guarantee that a full layer will be completed before the spot instance is reclaimed and shut down[3]. In such a scenario, the entire computation

[3]Indeed, in our testing, we show this to be the case. See Section 7.

```
let mut sxy_wires = Vec::new();
// Compute PEU 1: Compute products for all SNPs
let mut all_products = Vec::new();
for snp_col in 0..X.cols {
    let mut snp_products = Vec::new();
    for row in 0..X.rows {
        let prod = builder.mul(y.idx(row, 0), X.idx(row, snp_col));
        snp_products.push(prod);
    }
    all_products.push(snp_products);
}
builder.handoff();
// Compute PEU 2: Sum products for each SNP
for snp_col in 0..X.cols {
    let mut dot_prod = all_products[snp_col][0];
    for i in 1..all_products[snp_col].len() {
        dot_prod = builder.add(dot_prod, all_products[snp_col][i]);
    }
    sxy_wires.push(dot_prod);
}
println!("Line 2 (sxy): {:?}", sxy_wires);
builder.handoff();
```

Listing 1: ThriftyMPC circuit snippet for simplified Cochran-Armitage test (lines 1 and 2 in Figure 5). The calls to `handoff()` splits the circuit into two explicit compute PEUs.

progress could be lost. To avoid such failures, we introduce a `handoff()` primitive to our circuit language, as shown in Listing 1. In our language, a `handoff()` call *splits* the current layer in half at that point. In effect, this breaks down computation into smaller units as necessary to support preemption tolerance. We refer to these units as *protocol execution units (PEUs)*.

Before execution, the coordinator parses an input circuit into PEUs. The coordinator first organizes the code into layers and creates *compute PEUs* based off of them. A compute PEU consists of the input wire indices, output wire indices, and arithmetic operation (addition, multiplication, or scalar operation) of each gate within a layer. In parsing the circuit, if it encounters a `handoff()` call, the coordinator splits the compute PEU at that point, creating two new ones.

During ThriftyMPC execution, the coordinator transmits each PEU to every spot instance concurrently for synchronous execution. After each compute PEU, the coordinator checks whether any compute servers have received preemption notices. If so, the coordinator initiates a handoff by sending a *handoff PEU* to the current committee. A handoff PEU consists of the identifiers of the current committee $\mathcal{S}_\ell$ and the next committee $\mathcal{S}_{\ell+1}$; its functionality is dependent on the preemption tolerance mechanism chosen (Section 7.2). If no instances have been preempted, the coordinator skips the handoff and sends the next compute PEU.

By splitting compute PEUs with `handoff()` calls, the coordinator will check more often for potential handoffs, providing manual control that helps adapt large-scale workloads to be preemption-friendly. In effect, our PEU approach modifies the layered circuit paradigm to better support the spot instance scenario[4].

4.3 Preemption Tolerance

MPC over spot instances requires preemption tolerance for the following reasons: (1) server dropouts may leave too few shares for reconstruction causing the protocol to fail and (2) server dropouts may introduce security risks if mechanisms to recover from these

[4]As an alternative approach, we will show in Section 7.2 that we also design preemption tolerance that does *not* require these modifications to function.

Migration (Round ℓ)
$\mathcal{S}_\ell \leftarrow$ current committee, $\mathcal{R}_\ell \leftarrow \emptyset$

Compute Phase
$\forall P \in \mathcal{S}_\ell$ (parallel): execute compute PEU_ℓ
If P receives reclamation notice: notify coordinator, $\mathcal{R}_\ell \leftarrow \mathcal{R}_\ell \cup \{P\}$
Upon completion: P notifies coordinator
  If: $\mathcal{R}_\ell \neq \emptyset$:
    for each $P \in \mathcal{R}_\ell$, coordinator selects replacement spot instance if available, else substitute
    set $\mathcal{S}_{\ell+1}$ accordingly and signal handoff
  Else: $\mathcal{S}_{\ell+1} \leftarrow \mathcal{S}_\ell$

Handoff Phase (if needed)
$\forall P \in \mathcal{S}_\ell$: execute handoff PEU_ℓ (reshare output wires to $\mathcal{S}_{\ell+1}$)
$\forall Q \in \mathcal{S}_{\ell+1}$: sum received shares

Advance Round
Proceed to round $\ell + 1$ with committee $\mathcal{S}_{\ell+1}$

Figure 2: Migration in ThriftyMPC.

Resharing: Round ℓ

For each party $P_i \in \mathcal{S}_\ell$:
  Let $\{w_j\}_j$ be the output wires of P_i
  For each wire w_j: compute fresh t-out-of-n resharing $\{w_j^{(k)}\}_{k=1}^n \leftarrow \text{Reshare}_t(w_j)$
  Send $w_j^{(k)}$ to corresponding party $P_k \in \mathcal{S}_{\ell+1}$

For each party $P_k \in \mathcal{S}_{\ell+1}$:
  For each wire w_j: reconstruct $w'_j \leftarrow \sum_{i=1}^{|\mathcal{S}_\ell|} w_j^{(i)}$
  Use w'_j as input to compute phase of round $\ell + 1$

Figure 3: Resharing step under migration.

dropouts regenerate or redistribute shares in a way that allows our adversary to obtain state beyond the tolerated corruption threshold. To prevent these failures and maintain security throughout the protocol execution, ThriftyMPC provides both *migration* and *checkpointing* preemption tolerance mechanisms, either of which can be chosen at the beginning of the protocol execution.

Migration. Under migration, each round consists of one or more compute phases terminated by a handoff phase, as shown in Figure 2. During round ℓ, if a server receives a preemption notice, it immediately notifies the coordinator of its pending revocation. At the same time, the server continues executing the current compute phase. The compute phase for ℓ ends once all servers in $\mathcal{S}_\ell$ notify the coordinator of completion. At this point, the coordinator selects a new committee $\mathcal{S}_{\ell+1}$ for the next round, replacing any instance that received a preemption notice with a newly provisioned server. Servers that have not been preempted during ℓ continue to be in $\mathcal{S}_{\ell+1}$; if no committee members ever receive a preemption notice, there is no change in committee membership.

Handoff. This phase consists of migrating in-memory computation state from $\mathcal{S}_\ell$ to $\mathcal{S}_{\ell+1}$, if any committee change has occurred. During handoff, protocol state is migrated by resharing the state collectively held by the servers in $\mathcal{S}_\ell$. Following the optimized degree-reduction technique of GRR [GRR98], each server in $\mathcal{S}_\ell$ generates a fresh t-out-of-n resharing of every output wire at the end of round ℓ. Each server distributes its reshared values to the corresponding members of the next committee $\mathcal{S}_{\ell+1}$. For each wire, servers in $\mathcal{S}_{\ell+1}$ locally combine the received shares (by summation) to reconstruct a valid sharing of the wire, which serves as input to the compute phase of round $\ell + 1$, as depicted in Figure 3.

After all servers of $\mathcal{S}_{\ell+1}$ have notified the coordinator of receipt of the resharings intended for them, the handoff phase of ℓ is complete. This protocol has the effect of "re-randomizing" state: servers in $\mathcal{S}_{\ell+1}$ have different shares than those in $\mathcal{S}_\ell$. This step is used by Fluid MPC [CGG+21] to establish security under churn; an adversary who has

access to data in both both $\mathcal{S}_\ell$ and $\mathcal{S}_{\ell+1}$ does not gain any additional information. We will discuss security further in Section 5.

Addressing the cloud setting. The above flow assumes the coordinator is able to acquire a new replacement spot instance for all reclaimed parties of the prior committee. Practically, this may not always be the case since spot market conditions fluctuate; pricing and spot-instance availability varies over time. As we mentioned in Section 3, real-time events do not always align with abstract round boundaries. In ThriftyMPC, we address this issue through the use of dedicated long-lived instances we term *substitutes*. Like spot instances, each client is associated with a substitute throughout the protocol execution. But, since we desire cost-efficiency, these substitutes are low-cost and low-compute so they can cheaply be run at on-demand prices. Substitutes are utilized only when migration is required and a replacement spot instance is not available for the next committee. Once the coordinator procures a replacement spot instance, the substitute migrates its in-memory state to this replacement during the following handoff. The substitute then is shut down, and a new substitute takes its place.[5]

Under migration, transfer of state needs to complete before spot instance preemption. Otherwise, the state held by the revoked spot instance is permanently lost since it is never persisted. Because migration assumes completion of a compute phase before transferring state to a replacement party, the compute phase needs to be short enough such that its completion, and the handoff of outputs, can be done before the preempted spot instance is revoked. This constraint highlights the importance of our aforementioned `handoff()` primitive.

Checkpointing. Under checkpointing, each round consists of a single compute phase and may also include one or multiple handoff phases as necessary. Unlike migration, we rely on dedicated, long-lived *data stores* to persist protocol state for preemption tolerance, as shown in Figure 4.

In ThriftyMPC, we initially checkpoint at the beginning of the first round of protocol execution. Afterwards, checkpoints occur once per round at the end of each compute phase, i.e., state checkpointed at the end of round ℓ serves as input to the compute phase of round $\ell + 1$. A round is considered complete once all servers notify the coordinator of successful checkpointing, which marks the transition to the next round. This approach is akin to journaling a disk write before executing it.

Handoff. If a spot instance receives a preemption notice during the compute phase of round ℓ, it immediately notifies the coordinator, halts computation, and can go offline. Upon receiving this notice, the coordinator signals the beginning of a handoff phase. The coordinator provisions a replacement spot instance, which retrieves the most recently checkpointed state from the data store associated with the revoked spot instance. At this point, the committee $\mathcal{S}_\ell$ has now changed, with the revoked member being replaced by the newly provisioned instance. The coordinator then signals the updated committee $\mathcal{S}_\ell$ to begin protocol execution from the start of round ℓ, i.e., retry the current compute phase. During round ℓ, if no preemption notices are received by the members of committee $\mathcal{S}_\ell$, no handoff phases are necessary. In this scenario, after completion of the compute phase, the coordinator simply signals to begin round $\ell + 1$. Once $\ell + 1$ begins, the servers can delete their checkpoints from ℓ to save storage space/cost.

Addressing the cloud setting. With checkpointing as the preemption tolerance strategy, we can recover from spot instance preemptions that occur at any time without permanently stalling progress. In addition, checkpointing can handle more computationally expensive PEUs since there is not a time constraint on state transfer. This means checkpointing can be a more aggressive approach to computation, where rounds are equivalent to the largest possible circuit layers, i.e., until multiplications are required.

[5] Refreshing the substitute helps us achieve stronger security (Section 5).

Checkpointing (Epoch ℓ)
$\mathcal{S}_\ell \leftarrow$ current committee
Compute Phase
$\forall P \in \mathcal{S}_\ell$: execute compute PEU_ℓ
If: P receives preemption notice: halt computation and notify coordinator
Else: P checkpoints local state to data store
Upon successful checkpointing P notifies coordinator
Handoff Phase
Coordinator provisions replacement spot instance P'
P' retrieves latest checkpoint from data store associated with P
Revoked party P in $\mathcal{S}_\ell$ is replaced with P'
Coordinator signals $\mathcal{S}'_\ell$ to retry compute phase of epoch ℓ
Advance Epoch
Proceed to round $\ell + 1$

Figure 4: Checkpointing in ThriftyMPC.

Because checkpointing does not utilize substitutes, protocol progression requires a new spot instance to be procured via the coordinator. As a result, checkpointing can be subject to pauses in protocol execution because procuring replacement servers is not instantaneous[6].

Another requirement of checkpointing is that persisted data must be retrievable by the server assigned to compute on that state, while remaining confidential from all other servers within a committee. Using a cloud-native object store (e.g., AWS S3, GCP Cloud Storage, OpenStack Swift) as our checkpoint data store, we can use traditional cloud access controls and key management to ensure that these stores are only accessible to servers operating on the client's behalf. We achieve this isolation through Service Accounts on GCP (see Section 4.1) that effectively restrict access to object storage. A newly procured instance is provisioned by the coordinator under the Service Account of the instance it is replacing.

5 Security Model

In this section, we formalize the adversary's capabilities and determine the strongest adversary that each preemption tolerance mechanism can tolerate. We first model adversarial capabilities both mechanisms can withstand, and then identify the specific threats that migration and checkpointing can tolerate respectively. Based on this analysis, we provide proofs of security in Appendix B.

As discussed in Section 3, we consider a semi-honest adversary model, as is common in our application space [MMD19]. Thus, we can withstand at most a cloud adversary $\mathcal{A}$ who may corrupt online committee members during the execution stage of the protocol to learn the local state of each corrupted party but cannot deviate from the protocol. Our underlying GRR MPC primitive [GRR98] requires we bound the number of corruptions an adversary can perform to a minority of committee members in a round to achieve information-theoretic security.

Our adversary $\mathcal{A}$ may corrupt online parties in any round during protocol execution, provided the corruption threshold for the current round is not exceeded. Additionally, within each round, we can also tolerate *adaptive* corruptions; $\mathcal{A}$ may choose when to corrupt a party at any point during the round, potentially based on information learned so far, e.g., observed network traffic patterns.

In our system, committee changes only occur when spot instances are preempted. This means that it is possible for an adversary to corrupt a spot instance in round ℓ which was also active in rounds $< \ell$. We term this *retroactive effect*, as $\mathcal{A}$ may learn the internal

[6] In Section 6 we examine the cost ramifications of this overhead in more detail.

state of a corrupted party from all prior rounds in which that party participated. In the cloud, this captures compromises that expose persistent storage, snapshots, or backups. Retroactive corruptions are allowed provided that the number of corrupted parties does not exceed the corruption threshold in the current or any prior round.

Migration. Traditional MPC protocols typically assume static corruptions, where the set of corrupted parties is set at the start of the protocol and does not change throughout execution [Kel20, BOGW88, GRR98]. In contrast, dynamic MPC considers long-running computations where the participating parties may change over time, and the adversary may corrupt different parties across rounds—potentially even corrupting every party over the course of the protocol [ELL22, BELO15, ELL20].

This raises the question of how corruption patterns across rounds can be handled. Proactive security [OY91, BELO14] addresses this by periodically refreshing protocol state so that each round is independent, ensuring that long-term security holds even under changing corruptions. Thus, leveraging proactive security, we can say that $\mathcal{A}$ may corrupt servers in a *non-fixed manner*: $\mathcal{A}$ may corrupt any active server in round ℓ, as long as the currently corrupted number of servers in that round does not exceed the corruption threshold.

Proactive security is a useful feature in the context of ThriftyMPC, because it means that we can withstand a stronger adversary; one that may exploit repeated spot preemptions to corrupt servers associated with different clients over time. Without proactive security, such an adversary would eventually be able to obtain enough shares from different clients across rounds to leak information and break security.

Recall that Fluid MPC [CGG+21] shows leveraging GRR's [GRR98] "re-randomization" optimization is one way of providing this security guarantee. Our migration mechanism (Section 4.3) incorporates this technique during handoff, and therefore can withstand the dynamic MPC adversary in Definition 1. We provide a proof in Appendix B with additional details.

Definition 1 (Dynamic MPC adversary)**.** We say that the $(\mathcal{A}, \mathcal{E})$ results in a *Dynamic MPC adversary* if $\mathcal{A}$ can corrupt the servers in an adaptive, non-fixed manner with retroactive effect. Specifically, in epoch ℓ, the environment $\mathcal{E}$ can adaptively select a set of servers to be corrupted, $\mathcal{T}_\ell \subseteq \mathcal{S}^\ell$, where $\mathcal{S}^\ell$ is the set of active servers in that epoch. Upon corrupting the server, $\mathcal{A}$ learns its entire past state. The set of servers that $\mathcal{A}$ can corrupt, and the corresponding retroactive effect, is determined by the corruption threshold $\tau < \frac{1}{2}$, specifying that $\forall \ell,\ |\mathcal{T}_\ell| < \tau \cdot |\mathcal{S}^\ell|$.

To support migration, as in previous work [SLG+15, WUG+17, WMLP20, GMT+22], we employ a coordinator node and substitutes. We are careful to design the coordinator such that it can also operate semi-honestly, as in prior work in (non-dynamic) helper-assisted MPC [KKP+24a]. The coordinator does not operate or view private data or shares from any party, and only exists to provide synchronization and continuity for all computing nodes. We envision the coordinator being jointly operated by all parties to the computation for accountability purposes. By designing our coordinator in this way, an adversary who corrupts the coordinator in addition to the parties does not improve the adversary's capability. When using substitutes for computation, we make sure to delete and refresh them after a new spot instance is available, so we can achieve proactive security. We discuss further in Appendix B.1.

Checkpointing. The adversary model and security of migration preemption tolerance mostly follows from the definition of dynamic MPC. Checkpointing, on the other hand, does not.

Interestingly, even though checkpointing can be understood as dynamism (i.e., it allows parties to enter and leave the computation), it does *not* achieve proactive security against the adversary in Definition 1. While reading and resuming from storage allows us to

continue the protocol and handle churn, it forces us to skip the "re-randomization" of state, and this in turn impacts security. Consider the case where the adversary $\mathcal{A}$ corrupts a node q at round ℓ, and q is reclaimed and replaced by q'. By assumption q' is no longer corrupted, so the adversary moves to corrupt another node z. But since state was not re-randomized, the adversary now has the state of q' (which is q) *in addition to* that of z. Re-running previously executed MPC protocol steps forces a change in how to think about the relationship between dynamism and security.

With this in mind, it does not seem possible to achieve proactive security under our checkpointing approach. Instead, we modify the corruption model; the adversary $\mathcal{A}$ can adaptively corrupt a set $\mathcal{T}_\ell$ of the servers in a *fixed* manner: once $\mathcal{A}$ reaches the corruption threshold, it can only corrupt a server if that server is a replacement for a server that was already corrupted earlier. We formalize this new adversary type by tweaking Definition 1 into Definition 2:

Definition 2 (Checkpoint-Dynamic MPC adversary)**.** We say that the pair $(\mathcal{A}, \mathcal{E})$ results in a *Checkpoint-Dynamic MPC adversary* $\mathcal{A}$ if the environment $\mathcal{E}$ can dynamically select a set $\mathcal{T}_\ell \subseteq \mathcal{S}^\ell$ of servers to be corrupted in an adaptive, fixed manner with retroactive effect. Upon corrupting the server, $\mathcal{A}$ learns its entire past state. The set of servers that $\mathcal{A}$ can corrupt is determined by the corruption threshold $\tau < \frac{1}{2}$, specifying that in epoch ℓ, $\forall \ell,\ |\mathcal{T}_\ell| < \tau \cdot |\mathcal{S}^\ell|$.

This corruption model still allows for churn of the *servers* (as in dynamic MPC), but ties them more strongly to the *clients*. While not as strong as the adversary discussed in Definition 1, it still captures a reasonable and relevant class of adversaries: the other parties in the computation, as in traditional MPC. Importantly, as we will see in Section 7, this relaxation in security results in (1) better support for large workloads that are slow to migrate and (2) concrete cost reductions which may be prioritized for those with a lower budget. As such, we believe the above definition, which has not been studied in the dynamic MPC literature, is of independent interest to dynamic MPC protocol designers. We provide a proof of security in Appendix B.

Malicious security. Although we consider the semi-honest setting because it is common in our application space [MMD19], we designed preemption-tolerance mechanisms that are agnostic to the choice of underlying Layered MPC primitive. As such, our preemption strategies can therefore inherit the security properties of the underlying scheme when appropriate defenses are applied. For instance, the malicious-security compiler used in Fluid MPC [CGG+21] could be applied to the secret sharings produced during our migration protocol, thereby extending malicious security guaranties to the migration process itself. A natural direction for future work is to formalize the conditions under which such compilers compose with our preemption tolerance mechanisms and to identify any additional assumptions required for end-to-end security.

6 Experiment Setup

The ultimate goal of ThriftyMPC is to provide significant cost savings for collaborative data science. We demonstrate these cost savings by evaluating ThriftyMPC in the context of GWAS. We perform a Cochran-Armitage trend test: a form of the chi-squared test that outputs the association between diseases/traits (i.e., phenotypes) and inherited genetic mutations, termed single nucleotide polymorphisms, or SNPs.

The association test is implemented as an MPC circuit via our circuit builder abstraction (see Section 4.2) and run on our ThriftyMPC framework using spot instances. We evaluate preemption tolerance under both migration and checkpointing, leveraging BGW [BOGW88] as our underlying MPC primitive. Notably, our evaluations of ThriftyMPC under migration can be viewed as a practical instantiation of Fluid MPC's [CGG+21] semi-honest setting

Protocol: $[r^2] \leftarrow$ CochranArmitageSimplified($[X], [y]$)
Require: Genotype matrix $X \in \{0,1,2\}^{n \times m}$, phenotype vector $y \in \{0,1\}^{n \times 1}$

/ Calculate summary statistics */*
1: $[s_x] \leftarrow \mathbf{1}_n^T[X]$
2: $[s_{xy}] \leftarrow [y]^T[X]$
3: $[s_{xx}] \leftarrow \mathbf{1}_n^T([X] \odot [X])$, where $\odot$ is element-wise product
4: $[s_y] \leftarrow [y]^T\mathbf{1}_n$
5: $[s_{yy}] \leftarrow [y]^T[y]$
/ Calculate test statistics */*
6: $[\bar{s}_y] \leftarrow \frac{1}{n}[s_y]$
7: $[\bar{s}_x] \leftarrow \frac{1}{n}[s_x]$
8: $[a] \leftarrow [s_{xy}] - n \cdot [\bar{s}_x][\bar{s}_y]$
9: $[b_1] \leftarrow [s_{yy}] - n \cdot [\bar{s}_y][\bar{s}_y]$
10: $[b_2] \leftarrow [s_{xx}] - n \cdot [\bar{s}_x] \odot [\bar{s}_x]$
11: $[\sqrt{b_1}], [1/\sqrt{b_1}] \leftarrow$ SqrtAndSqrtInverse($[b_1]$)
12: $[\sqrt{b_2}], [1/\sqrt{b_2}] \leftarrow$ SqrtAndSqrtInverse($[b_2]$)
13: $[1/\sqrt{b}] \leftarrow [1/\sqrt{b_1}] \odot [1/\sqrt{b_2}]$
14: $[r] \leftarrow [a] \odot [1/\sqrt{b}]$
15: $[r^2] \leftarrow [r] \odot [r]$
16: **return** $[r^2]$

Figure 5: Simplified Cochran-Armitage Trend Test (no covariate correction). Lines 11 and 12 are run in parallel.

adapted to spot-instance execution. For a baseline comparison, this test is also implemented and run on traditional on-demand instances using the shamir-party.x protocol of MP-SPDZ [Kel20], a state-of-the-art MPC framework. This baseline is also grounded in BGW-style secret sharing, enabling a consistent comparison while leveraging MP-SPDZ's optimized preprocessing for the online phase.

Association testing involves linear regression operations that are computationally expensive to compute in traditional arithmetic circuits using multiplications and additions. While there are domain specific MPC protocols for more efficiently computing these tests [CWB18b], our evaluations show that even non-domain specific MPC implementations can realize significant cost savings on the cloud through ThriftyMPC's preemption tolerance mechanisms.

Simulating realistic GWAS. We use simulated GWAS data rather than real genomic datasets, which require institutional approval and strict data access controls that would limit reproducibility. Simulated data matching realistic study dimensions allows us to evaluate performance while ensuring our experiments can be independently verified and ethically responsible.

In a GWAS study, each client provides genomic information describing SNPs at a genomic index. Following the data representation in [CWB18b], we model each study participant P_i's data as:

- **Genotype** $g_i^{AA}, g_i^{Aa}, g_i^{aa} \in \{0,1,2\}^m$: indicators for whether the jth SNP is homozygous-reference, heterozygous, or homozygous-alternative allele, respectively. This indicates whether the genetic mutation was not present (0), inherited from one parent (1), and inherited from both parents (2).
- **Phenotype** $y_i \in \{0,1\}$: the binary trait of interest (e.g., disease status) for P_i

We initialize and populate the genotype and phenotype data structures with randomly generated values matching realistic GWAS dataset dimensions. Our dataset sizes are based on existing studies such as the lung cancer susceptibility study of never-smoking Asian women [dbG26, CWB18b], which comprised 9,178 samples (the number of individuals who are a part of the study) and 612,794 SNPs (per individual). Our simulations use matrices with similar dimensions: 9,000 samples and 600,000 SNPs. Because association tests can be performed independently across SNPs, we analyze the data in batches. Each batch processes 10 SNPs per individual, resulting in batch dimensions of 9,000 samples $\times$ 10

SNPs.

GWAS computation. We implement the Cochran-Armitage trend test following [CWB18b]. Our implementation excludes the covariate correction steps (lines 1-10 of Protocol 33 in [CWB18b]), as these can typically be performed as preprocessing steps and amortized across multiple association tests in real GWAS pipelines. We focus our evaluation on the core association test computation (lines 11-31), which includes calculating summary statistics and test statistics for each SNP.

Implementation and testbed. Our evaluations were run live on the Google Cloud Platform. The procurement costs reflect cloud pricing as of January 2026. The coordinator node ran on a traditional on-demand c2-standard-4 instance with 4 vCPUs and 16 GiB of memory, obtained at 0.17 USD per hour. Computing party nodes ran on c2-standard-30 spot instances, each with 30 vCPUs and 120 GiB of memory. Spot-provisioned compute nodes were obtained at 0.46 USD per hour, with their traditionally provisioned counterparts at 1.76 USD per hour. This is a spot price discount of approximately 74%. For migration, substitute nodes ran on traditional on-demand e2-standard-2 instances with 2 vCPUs and 8 GiB of memory, each obtained at 0.078 USD per hour. All GCP instances were located in data center region `us-east4`. For checkpointing, we utilized Google Object Storage.

We introduce notation here and throughout the next section as needed to support our cost analysis. For our evaluations, we set the number of parties, P, in a ThriftyMPC committee to 3 and we utilize a 2-out-of-3 threshold secret sharing scheme. For migration, we also set the number of substitutes, U, to 3. Our complete implementation comprises approximately 12,000 lines of Rust code. To our knowledge, this is the first practical, end-to-end implementation of dynamic MPC in general.

7 Evaluation

To quantify cost savings, our evaluation considers three dimensions. First, we measure the runtime of ThriftyMPC without preemption tolerance and compare it to a baseline MP-SPDZ deployment. This allows us to approximate how long a GWAS association test would take under an instantiation of baseline Layered MPC versus a traditional MPC framework. Next, we evaluate the deployment of our preemption-tolerance mechanisms within ThriftyMPC, assessing how effectively each mechanism handles spot instance churn. Finally, we use these cumulative findings to estimate the cloud cost savings achievable by ThriftyMPC when leveraging spot instances.

7.1 Runtime Performance

To evaluate how quickly ThriftyMPC completes association testing (Figure 5), we ran each association test circuit for at least 10 iterations on committees comprised of spot instances where no revocations took place. These results can be seen in Figure 6. ThriftyMPC's wall-clock runtimes appear to scale quadratically with increases in the sample sizes. The largest sample size we tested, 9,000, takes approximately 47.9 seconds. To put this into context, completing the entire GWAS test over all 600,000 SNPs and 9,000 samples would take ThriftyMPC approximately 33 days.

We evaluated the same association test in MP-SPDZ, a modern baseline for practical MPC [Kel20]. Both systems were benchmarked under similar conditions: c2-standard-30 instances, sample sizes ranging from 1,000 to 9,000, and a minimum of 10 iterations per configuration.

As shown in Figure 6, MP-SPDZ exhibits linear scaling with input size. MP-SPDZ's optimized implementation and use of offline preprocessing achieve approximately 2× faster wall-clock times compared to ThriftyMPC. For 9,000 samples processing 600,000 SNPs, MP-SPDZ would complete the full GWAS test in approximately 15 days.

While MP-SPDZ is more optimized than ThriftyMPC, we show in Section 7.3 that ThriftyMPC and its preemption-tolerant use of spot instances yields cost savings that can even outweigh MP-SPDZ's 2× runtime advantage.

7.2 Preemption Tolerance

Successful preemption tolerance involves both moving state to replacement compute, which incurs overhead, and acquiring replacement instances that can immediately join the computation. We evaluate both aspects below.

Overhead of preemption tolerance. We measure this overhead by manually triggering spot instance shutdowns during an ongoing computation. To study how handoff overhead scales with state size, we vary the size of the migrated state by executing computations over different sample sizes.

Migration. In migration, during a handoff, committee $\mathcal{S}_\ell$ transfers its entire protocol state to $\mathcal{S}_{\ell+1}$. We see in Figure 7 that even at our largest sample size, the time to complete a handoff (which we denote as δ) is still relatively short, taking a little over 2.5 seconds for state to be transferred from $\mathcal{S}_\ell$ to $\mathcal{S}_{\ell+1}$, scaling linearly with the sample size.

After a handoff, a substitute might be part of committee $\mathcal{S}_{\ell+1}$. When this is the case, the substitute's lower computational capacity introduces a performance bottleneck, resulting in longer batch execution times. To quantify this overhead, we execute our largest batch configuration (9,000 × 10 samples) using a committee consisting of two spot instances and one substitute, running the experiment for at least ten iterations. On average, these runs complete in approximately 65.54 seconds. Generalizing across sample sizes, the presence of a substitute increases overall batch runtime by approximately 27%; we denote this overhead as α.

Checkpointing. Under checkpointing, parties periodically persist their local state during computation. When replacement parties join, they retrieve the most recent checkpoint and resume execution from that state. We measure the cost of checkpointing by timing both parts of the checkpointing operation – save and load – at each sample size.

While we see in Figure 7 there is generally some runtime increase in checkpointing operations as sample sizes get larger, these operations are quick. For our experiments, we can assume a maximum checkpointing delay δ of 2 seconds for both writing to a checkpoint store and reading from it respectively. Overall, both migration and checkpointing incur minimal handoff overhead, accommodating preemption without introducing significant wall-clock time.

Replacing preempted parties. In addition to the actual handoff process, we also consider the time it takes to procure a replacement instance. We measure this by using GCP's REST API to provision new spot instances, from instance boot to the completion of a custom startup script, which simulates the time required for a fresh instance to become operational. In our testing, on average, this process takes 45.8 seconds with a standard deviation of 3.5 seconds.

Migration. GCP provides a 30-second revocation window between when a preemption notice is issued and when a spot instance is reclaimed. This window poses a constraint for migration, as a new spot instance cannot be reliably provisioned within this timeframe. Under migration this means all state for this computation would be lost. This limitation confirms the need for substitutes in the next committee (see Section 4.3) when using migration.[7]

The 30-second window also constrains how circuits can be structured for migration. Recall that migration assumes there is always enough time for a committee to complete

[7]Revocation windows differ across cloud service providers. For example, AWS emits a preemption notice 2 minutes in advance [AWS26], which means substitutes may not be required, leading to additional savings for ThriftyMPC.

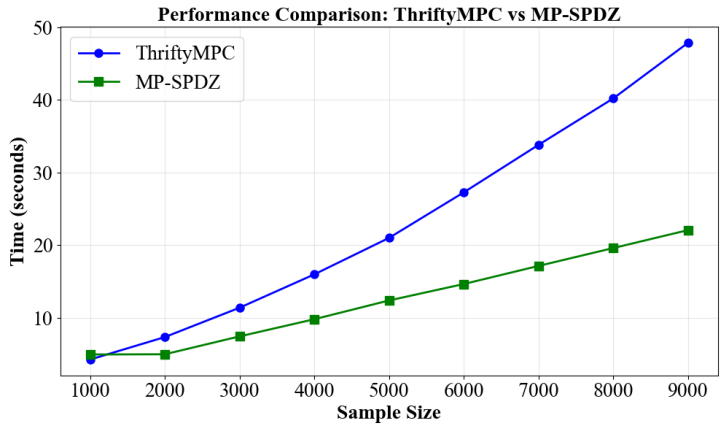


Figure 6: Comparing performancce of ThriftyMPC without preemption tolerance and MP-SPDZ.

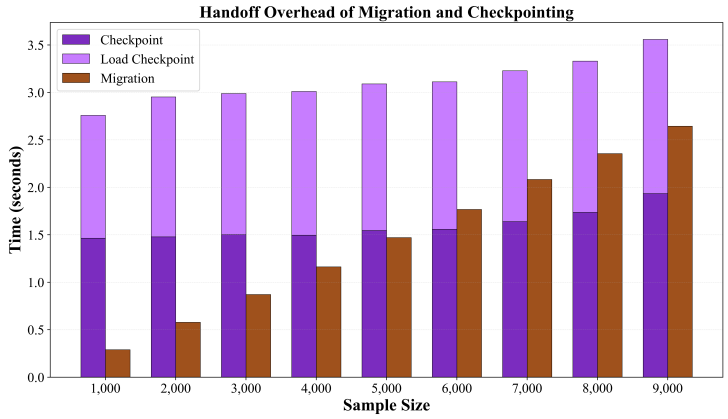


Figure 7: Comparison of handoff operation running times of migration and checkpointing

both a compute PEU *and* a handoff PEU (if necessary), i.e., compute PEUs must be fast enough to fit within GCP's 30-second window while still leaving sufficient time for a handoff. Based on Figures 6 and 7, batches with sample sizes of 6,000 or smaller can be executed as a single compute PEU, with enough time for a subsequent handoff. For larger batches, compute PEUs must be divided into smaller units, with handoffs inserted between them in a layered circuit structure. In practice, this means that circuit design requires careful consideration to make sure that each layer of the circuit is computationally inexpensive enough to fit within a revocation window. Our circuit builder (Section 4.2) supports this approach with its `handoff()` primitive.

Checkpointing. In contrast, checkpointing is not sensitive to the revocation window. As long as state has been successfully persisted, execution can be resumed on a subsequent instance, whenever the instance is procured. As a result, checkpointing can support more computationally expensive compute PEUs. Storing the checkpoint is the only requirement to accommodate preemption, and Figure 7 shows that the time to do this is much smaller than the revocation window even at higher sample sizes.

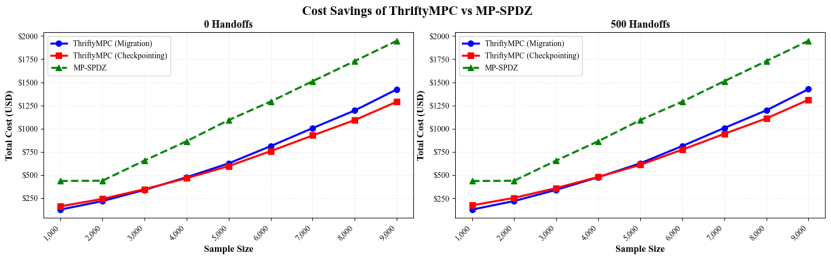


Figure 8: Cost Comparison of ThriftyMPC's leveraging preemption tolerance and MP-SPDZ

7.3 Cost Savings

With our benchmarks, we can estimate the cost for a ThriftyMPC execution leveraging preemption tolerance given we have the following information: the computation's runtime under ThriftyMPC and the associated infrastructure costs for the chosen preemption tolerance mechanism[8].

Migration. Incorporating the notation used in Section 6 and Section 7.2, the total runtime of ThriftyMPC (in hours) under migration can be modeled by the following:

$$RTime_{\text{total}} = \frac{(N - n_{\text{sub}})k + n_{\text{sub}}k(1+\alpha) + h\delta}{3600}$$

We introduce N to denote the number of computation batches, each taking k seconds to complete, with h denoting the number of handoffs. n_{sub} represents the number of batches executed where a substitute contributed to computation. We set this to half the total number of handoffs to model replacing substitutes as often as possible (i.e., in the next committee). The first term accounts for tasks run on spot instance compute, the second term accounts for tasks run with the use of substitutes, and the third term accounts for handoff latency. We can then model the total cost of migration:

$$RCost_{\text{total}} = RTime_{\text{total}}\Big(P \cdot c_{\text{spot}} + c_{\text{coordinator}} + U \cdot c_{\text{sub}}\Big)$$

where c_{spot}, $c_{\text{controller}}$, and c_{sub} are the hourly costs of a spot instance, coordinator instance, and substitute instance, respectively.

In Figure 8, we display the cost of running ThriftyMPC under migration and the cost of MP-SDPZ on-demand (i.e., without preemption tolerance) using the test setup described in Section 6. We see that even accounting for the overhead of our premeption-tolerance mechanisms, and the fact that MP-SPDZ has a 2× runtime advantage, the cost savings are substantial. For small batches, ThriftyMPC's computational performance is similar to MP-SPDZ, yielding up to 70% savings (e.g., sample size of 1000). As sample sizes increase and MP-SPDZ optimizations become more effective, the relative savings gradually decrease. However, even at our largest tested sample size, where the advantages of MP-SDPZ's optimizations are most clearly seen, ThriftyMPC still achieves roughly 27% savings.

Checkpointing. The total runtime of ThriftyMPC (in hours) under checkpointing can be modeled by the following:

$$CTime_{\text{total}} = \frac{N(k+\delta) + h(\delta + \theta + k)}{3600}$$

where variables are as defined previously. We introduce θ to represent the average time to procure a new spot instance. We account for θ (set to 45 seconds) because checkpointing

[8] In GCP, communication between nodes in the same zone is free; our nodes all reside in the same zone, so communication cost is not a factor [GCP26b].

does not use substitutes and therefore the protocol needs to pause until a new replacement party can join. At the same time, not requiring substitutes means we do not need to account for the computational latency they introduce. In this equation, hk represents re-running a round after a handoff.

To model cost, we introduce D to represent the required data storage operations. Under checkpointing, each spot instance is associated with a cloud storage bucket used to persist its local state. GCP's Standard Cloud storage rate in region `us-east4` $c_{data} = 0.000017808$ USD per GB per hour. While Figure 9 shows individual checkpoints are small, and we can overwrite prior checkpoints for further cost savings, our workload performs frequent checkpointing and incurs associated storage operation costs.

GCP charges separately for storage capacity and object storage operations, with different operations (read, write) categorized and charged differently. Precisely modeling these costs across providers is complex due to differences in pricing models, request types, and metadata overheads. To simplify and ensure robustness, we approximate storage costs using a global storage budget D, which we set to 100 GB. This budget intentionally overestimates storage usage, absorbing unmodeled costs like frequent write operations, metadata overhead, and provider-specific billing nuances. Our storage cost estimates should be interpreted as an upper bounds. We model the total cost of checkpointing as:

$$CCost_{\text{total}} = CTime_{\text{total}}\Big(P \cdot c_{\text{spot}} + c_{\text{coordinator}} + D \cdot c_{\text{data}}\Big)$$

where c_{spot}, $c_{\text{controller}}$, and c_{data} are the hourly costs of a spot instance, coordinator instance, and data storage, respectively.

The cost of running ThriftyMPC under checkpointing and cost of MP-SPDZ on-demand is also displayed in Figure 8. We see that even though checkpointing requires pausing execution to procure replacement parties and re-running rounds when handoff occurs, its lack of substitute usage results in more savings compared to migration at larger sample sizes. Data storage is cheap so utilizing cloud storage does not significantly increase cost. On the other hand, periodic state persistence leads to increases in runtime which make checkpointing slightly more expensive at smaller sample sizes. Similar to migration, significant savings (around 60% in comparison to MP-SDPZ) are realized at smaller sample sizes like 1000. At our largest sample size, checkpointing outperforms migration in terms of cost effectiveness: savings of $\approx 31\%$ when compared to MP-SPDZ.

Auxiliary communication costs. Communication within GCP is not always charged and we take advantage of free intra-zone communication via internal IP addresses [GCP26b]. We evaluate less favorable scenarios where communication is billed in Appendix A.

Migration vs checkpointing. The choice of preemption-tolerance mechanism depends on client preferences for cost, flexibility, and security. Across most sample sizes, checkpointing yields better cost savings than migration. This difference stems from the infrastructure requirements of each mechanism. Migration requires substitute compute instances to prevent loss of state when new spot instances are unavailable. Checkpointing only requires persistent cloud storage, which is substantially cheaper than compute, especially for longer running computations. Moreover, checkpointing offers greater flexibility in circuit design, tolerating more computationally expensive PEUs. Migration, by contrast, requires that a compute and handoff PEU be able to complete within the revocation window, which constrains the runtime of individual PEUs.

For computations that require stronger security guaranties, the migration model might be necessary. This model protects against a stronger adversary than checkpointing, one who is free to corrupt servers across rounds in a non-fixed manner. But, this comes at greater circuit design costs (i.e., to add `handoff()` appropriately) and greater monetary costs at larger batches. In other words, more security means paying a little extra: a common trope.

8 Conclusion

We present ThriftyMPC, a cost-effective approach to large-scale MPC computation on the cloud. ThriftyMPC's novel preemption-tolerance mechanisms enable secure computation over spot instances. We show significant cost savings reaching as high as 70% over the traditional state-of-the art MPC framework, MP-SPDZ, while maintaining provable security guaranties.

Acknowledgments

This work is supported by National Science Foundation (NSF) awards 2450539 and 2451597, a PSC-CUNY award, and a Google Cyber NYC award. This work also used Jetstream2 computing resources through allocations from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by NSF awards 2138259, 2138286, 2138307, 2137603, and 2138296.

Parts of this paper were edited for grammar using Overleaf's WriteFull and ChatGPT. Also, ChatGPT was used to assist in editing charts for figures. All ideas, data, analysis, and experimental results were produced by the authors.

References

[ACC+18] Abdelrahaman Aly, Karl Cong, Daniele Cozzo, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Oliver Scherer, Peter Scholl, Nigel P Smart, Titouan Tanguy, et al. SCALE-MAMBA, 2018.

[AHKP22] Anasuya Acharya, Carmit Hazay, Vladimir Kolesnikov, and Manoj Prabhakaran. SCALES - MPC with small clients and larger ephemeral servers. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 502–531. Springer, Cham, November 2022. doi:10.1007/978-3-031-22365-5_18.

[Ale] Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. https://github.com/aicis/fresco.

[AWS25] AWS. Spot Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html, December 2025. Online; accessed 23 Dec 2025.

[AWS26] AWS. Spot instance interruption notices. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html, 2026. Accessed: 2026-02-02.

[BBG+21] Fabrice Benhamouda, Elette Boyle, Niv Gilboa, Shai Halevi, Yuval Ishai, and Ariel Nof. Generalized pseudorandom secret sharing and efficient straggler-resilient secure computation. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part II*, volume 13043 of *LNCS*, pages 129–161. Springer, Cham, November 2021. doi:10.1007/978-3-030-90453-1_5.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343.

Springer, Berlin, Heidelberg, February 2009. doi:10.1007/978-3-642-03549-4_20.

[BDST22] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. Motion–a framework for mixed-protocol multi-party computation. *ACM Transactions on Privacy and Security*, 25(2):1–35, 2022.

[BELO14] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. How to withstand mobile virus attacks, revisited. In Magnús M. Halldórsson and Shlomi Dolev, editors, *33rd ACM PODC*, pages 293–302. ACM, July 2014. doi:10.1145/2611462.2611474.

[BELO15] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 2015*, volume 9092 of *LNCS*, pages 23–41. Springer, Cham, June 2015. doi:10.1007/978-3-319-28166-7_2.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988. doi:10.1145/62212.62213.

[BHK+23] Lennart Braun, Moritz Huppert, Nora Khayata, Thomas Schneider, and Oleksandr Tkachenko. FUSE - flexible file format and intermediate representation for secure multi-party computation. In Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik, editors, *ASIACCS 23*, pages 649–663. ACM Press, July 2023. doi:10.1145/3579856.3590340.

[BHS+17] Chet Birger, Megan Hanna, Edward Salinas, Jason Neff, Gordon Saksena, Dimitri Livitz, Daniel Rosebrock, Chip Stewart, Ignaty Leshchiner, Alexander Baumann, et al. Firecloud, a scalable cloud-based platform for collaborative genome analysis: strategies for reducing and controlling costs. *bioRxiv*, page 209494, 2017.

[BJMS20] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure MPC: Laziness leads to GOD. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 120–150. Springer, Cham, December 2020. doi:10.1007/978-3-030-64840-4_5.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Berlin, Heidelberg, October 2008. doi:10.1007/978-3-540-88313-5_13.

[BMIMT+23] Marta Belles-Munoz, Miguel Isabel, Jose Luis Munoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A Circuit Description Language for Building Zero-Knowledge Applications . *IEEE Transactions on Dependable and Secure Computing*, 20(06):4733–4751, November 2023. URL: https://doi.ieeecomputersociety.org/10.1109/TDSC.2022.3232813, doi:10.1109/TDSC.2022.3232813.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 1–10, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/62212.62213.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (abstract) (informal contribution). In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, page 462. Springer, Berlin, Heidelberg, August 1988. doi:10.1007/3-540-48184-2_43.

[CCPS19] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'19, page 81–92, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3338466.3358922.

[CDGK22] Ignacio Cascudo, Bernardo David, Lydia Garms, and Anders Konring. YOLO YOSO: Fast and simple encryption and secret sharing in the YOSO model. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part I*, volume 13791 of *LNCS*, pages 651–680. Springer, Cham, December 2022. doi:10.1007/978-3-031-22963-3_22.

[CGG+21] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 94–123, Virtual Event, August 2021. Springer, Cham. doi:10.1007/978-3-030-84245-1_4.

[CWB18a] Hyunghoon Cho, David J Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology*, 36(6):547–551, 2018.

[CWB18b] Hyunghoon Cho, David J Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology*, 36(6):547–551, 2018. doi:10.1038/nbt.4108.

[CWR+20] Elizabeth T Cirulli, Simon White, Robert W Read, Gai Elhanan, William J Metcalf, Francisco Tanudjaja, Donna M Fath, Efren Sandoval, Magnus Isaksson, Karen A Schlauch, et al. Genome-wide rare variant analysis for thousands of phenotypes in over 70,000 exomes from two cohorts. *Nature communications*, 11(1):542, 2020.

[dbG26] dbGAP. Genome-Wide Association Study of Lung Cancer Susceptibility in Never-Smoking Women in Asia. https://dbgap.ncbi.nlm.nih.gov/beta/study/phs000716.v1.p1/#analyses, January 2026. Online; accessed 22 Jan 2026.

[DDG+23] Bernardo David, Giovanni Deligios, Aarushi Goel, Yuval Ishai, Anders Konring, Eyal Kushilevitz, Chen-Da Liu-Zhang, and Varun Narayanan. Perfect MPC over layered graphs. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part I*, volume 14081 of *LNCS*, pages 360–392. Springer, Cham, August 2023. doi:10.1007/978-3-031-38557-5_12.

[DEK21] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2183–2200. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov.

[DEP21] Ivan Damgård, Daniel Escudero, and Antigoni Polychroniadou. Phoenix: Secure computation in an unstable network with dropouts and comebacks.

Cryptology ePrint Archive, Report 2021/1376, 2021. URL: https://eprint.iacr.org/2021/1376.

[DIK10] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Berlin, Heidelberg, May / June 2010. doi:10.1007/978-3-642-13190-5_23.

[DKLZN24] Giovanni Deligios, Anders Konring, Chen-Da Liu-Zhang, and Varun Narayanan. Statistical layered MPC. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024, Part IV*, volume 15367 of *LNCS*, pages 362–394. Springer, Cham, December 2024. doi:10.1007/978-3-031-78023-3_12.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015. doi:10.14722/ndss.2015.23113.

[ELL20] Karim Eldefrawy, Tancrède Lepoint, and Antonin Leroux. Communication-efficient proactive secret sharing for dynamic groups with dishonest majorities. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *ACNS 2020, Part I*, volume 12146 of *LNCS*, pages 3–23. Springer, Cham, October 2020. doi:10.1007/978-3-030-57808-4_1.

[ELL22] Karim Eldefrawy, Tancrède Lepoint, and Antonin Leroux. Communication-efficient proactive MPC for dynamic groups with dishonest majorities. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 2022*, volume 13269 of *LNCS*, pages 565–584. Springer, Cham, June 2022. doi:10.1007/978-3-031-09234-3_28.

[EMP26] Daniel Escudero, Elisaweta Masserova, and Antigoni Polychroniadou. Towards scalable yoso mpc via packed secret-sharing. In Goichiro Hanaoka and Bo-Yin Yang, editors, *Advances in Cryptology – ASIACRYPT 2025*, pages 68–101, Singapore, 2026. Springer Nature Singapore.

[GCP26a] GCP. Spot VMs. https://docs.cloud.google.com/compute/docs/instances/spot, January 2026. Online; accessed 08 Jan 2026.

[GCP26b] GCP. Virtual private cloud pricing. https://cloud.google.com/vpc/pricing?hl=en#egress-within-gcp, 2026. Accessed: 2026-04-14.

[GG16] Andra Gumbus and Frances Grodzinsky. Era of big data: Danger of descrimination. *ACM SIGCAS Computers and Society*, 45(3):118–125, 2016.

[GHK+21] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 64–93, Virtual Event, August 2021. Springer, Cham. doi:10.1007/978-3-030-84245-1_3.

[GMT+22] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. Cocktail: A multidimensional optimization for model serving in cloud. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1041–1057, 2022.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987. doi:10.1145/28395.28420.

[GPS19] Yue Guo, Rafael Pass, and Elaine Shi. Synchronous, with a chance of partition tolerance. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 499–529. Springer, Cham, August 2019. doi:10.1007/978-3-030-26948-7_18.

[GRR98] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *17th ACM PODC*, pages 101–111. ACM, June / July 1998. doi:10.1145/277697.277716.

[Her23] Jonathan Herzog. Share and query encrypted data in AWS Clean Rooms. AWS Security Blog, 2023.

[HFKV12] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ANSI C. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 772–783. ACM Press, October 2012. doi:10.1145/2382196.2382278.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy*, pages 1220–1237. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00028.

[IPP+22] Yuval Ishai, Arpita Patra, Sikhar Patranabis, Divya Ravi, and Akshayaram Srinivasan. Fully-secure MPC with minimal trust. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 470–501. Springer, Cham, November 2022. doi:10.1007/978-3-031-22365-5_17.

[KDNGS+20] Ines Krissaane, Carlos De Niz, Alba Gutiérrez-Sacristán, Gabor Korodi, Nneka Ede, Ranjay Kumar, Jessica Lyons, Arjun Manrai, Chirag Patel, Isaac Kohane, et al. Scalability and cost-effectiveness analysis of whole genome-wide association studies on google cloud platform and amazon web services. *Journal of the American Medical Informatics Association*, 27(9):1425–1430, 2020.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1575–1590. ACM Press, November 2020. doi:10.1145/3372297.3417872.

[KH20] Niklas Krumm and Noah Hoffman. Practical estimation of cloud storage costs for clinical genomic data. *Practical Laboratory Medicine*, 21:e00168, 2020.

[KK07] Jonathan Katz and Chiu-Yuen Koo. Round-efficient secure computation in point-to-point networks. In Moni Naor, editor, *EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 311–328. Springer, Berlin, Heidelberg, May 2007. doi:10.1007/978-3-540-72540-4_18.

[KKP+24a] Banashri Karmakar, Nishat Koti, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. Asterisk: Super-fast mpc with a friend. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 542–560. IEEE, 2024.

[KKP+24b] Patrick Tser Jern Kon, Sina Kamali, Jinyu Pei, Diogo Barradas, Ang Chen, Micah Sherr, and Moti Yung. Spotproxy: Rediscovering the cloud for censorship circumvention. In *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[KPPS21] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 2651–2668. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/koti.

[KPRS22] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4PC for secure training and inference. In *NDSS 2022*. The Internet Society, April 2022.

[KSC23] Kevin Klassen, Jake Skinner, and Joel Coffman. Is securing an untrusted cloud with cryptography still economically unviable? In *Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing*, pages 1–6, 2023.

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013. doi:10.1145/2508859.2516744.

[LJA+18] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, pages 1–5, 2018.

[LN18] Ben Langmead and Abhinav Nellore. Cloud computing for genomic data analysis and collaboration. *Nature Reviews Genetics*, 19(4):208–219, 2018.

[LRK+18] Harris A Lewin, Gene E Robinson, W John Kress, William J Baker, Jonathan Coddington, Keith A Crandall, Richard Durbin, Scott V Edwards, Félix Forest, M Thomas P Gilbert, et al. Earth biogenome project: Sequencing life for the future of life. *Proceedings of the National Academy of Sciences*, 115(17):4325–4333, 2018.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015. doi:10.1109/SP.2015.29.

[MCPT24] Christian Mouchet, Sylvain Chatel, Apostolos Pyrgelis, and Carmela Troncoso. Helium: Scalable mpc among lightweight participants and under churn. Cryptology ePrint Archive, Report 2024/194, 2024. https://eprint.iacr.org/2024/194.

[MGC+16] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 112–127. IEEE, 2016.

[MK22] Philipp Muth and Stefan Katzenbeisser. Assisted MPC. Cryptology ePrint Archive, Report 2022/1453, 2022. URL: https://eprint.iacr.org/2022/1453.

[MMD19] Alexandros Mittos, Bradley Malin, and Emiliano De Cristofaro. Systematizing genome privacy research: A privacy-enhancing technologies perspective. *PoPETs*, 2019(1):87–107, January 2019. doi:10.2478/popets-2019-0006.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In Matt Blaze, editor, *USENIX Security 2004*, pages 287–302. USENIX Association, August 2004. URL: http://www.usenix.org/publications/library/proceedings/sec04/tech/malkhi.html.

[MR19] Melinda C Mills and Charles Rahal. A scientometric review of genome-wide association studies. *Communications biology*, 2(1):1–11, 2019.

[MSJ14] Ishai Menache, Ohad Shamir, and Navendu Jain. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 177–187, 2014.

[Mul] Multiparty.org Development Team. JavaScript Implementation of Federated Functionalities. https://github.com/multiparty/jiff.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017. doi:10.1109/SP.2017.12.

[OY91] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In Luigi Logrippo, editor, *10th ACM PODC*, pages 51–59. ACM, August 1991. doi:10.1145/112600.112605.

[Pas17] Mario Pascalev. Privacy exchanges: restoring consent in privacy self-management. *Ethics and Information Technology*, 19(1):39–48, 2017.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 2009*, pages 199–212. ACM Press, November 2009. doi:10.1145/1653662.1653687.

[Sch18] Berry Schoenmakers. Mpyc—python package for secure multiparty computation. In *Workshop on the Theory and Practice of MPC. https://github.com/lschoe/mpyc*, 2018.

[SD19] Jeffrey S Saltz and Neil Dewar. Data science ethical considerations: a systematic literature review and proposed project framework. *Ethics and Information Technology*, 21:197–208, 2019.

[SGS+15] Supreeth Subramanya, Tian Guo, Prateek Sharma, David Irwin, and Prashant Shenoy. Spoton: a batch computing service for the spot market. In *Proceedings of the sixth ACM symposium on cloud computing*, pages 329–341, 2015.

[Sha79a] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. doi:10.1145/359168.359176.

[Sha79b] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[SLG+15] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[TUS11] Byung Chul Tak, Bhuvan Urgaonkar, and Anand Sivasubramaniam. To move or not to move: The economics of cloud computing. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*, 2011.

[VZRS15] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M. Swift. A placement vulnerability study in multi-tenant public clouds. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 913–928. USENIX Association, August 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/varadarajan.

[WMK16] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. Emp-toolkit: Efficient multiparty computation toolkit, 2016.

[WMLP20] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. Spotnik: Designing distributed machine learning for transient cloud resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[WUG+17] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 620–634, 2017.

[XSDW16] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[XWW15] Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 929–944. USENIX Association, August 2015. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/xu.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986. doi:10.1109/SFCS.1986.25.

[ZE15] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. URL: https://eprint.iacr.org/2015/1153.

[Zgo23] Marcin Zgola. The rise of cloud repatriation: Why companies are bringing data in-house. *Forbes*, 2023.

[ZMFT24] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Everywhere all at once: Co-location attacks on public cloud faas. In *Proceedings of the 29th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, Volume 1, ASPLOS '24, page 133–149, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3617232.3624867.

[ZPGS19] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *2019 IEEE Symposium on Security and Privacy*, pages 724–738. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00045.

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 813–826. ACM Press, November 2013. doi:10.1145/2508859.2516752.

A Communication Costs

In our experiments, to further optimize costs, we deployed all instances and object storage within the same GCP region, enabling free intra-zone communication via internal IP addresses [GCP26b]. Because spot markets are volatile, however, maintaining strict intra-zone placement is not always guaranteed for the entire execution. To evaluate a less favorable deployment scenario, we also model cross-region communication costs within North America, where data transfer is billed at $0.02 per GB. In this setting, ingress into GCP remains free [GCP26b]. While these costs are small, we include them for completeness.

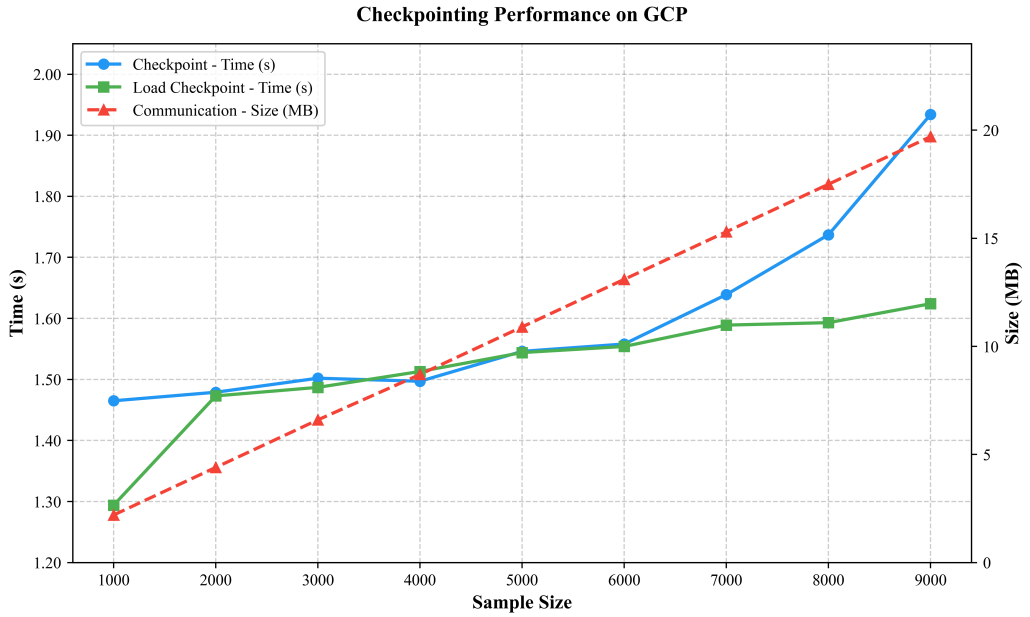


Figure 9: Understanding the overhead of checkpointing on GCP.

At our largest sample size tested (9,000), ThriftyMPC transfers approximately 28.7 KB of communication per batch between the coordinator and compute nodes, corresponding to 1.72 GB of aggregate data transfer across the full protocol execution. Under cross-region deployment, this results in an additional communication cost of approximately $0.034.

Unlike migration, checkpointing can introduce significant communication costs under cross-region deployments since state is persisted regardless of whether or not preemptions have occurred. At our largest sample size of 9,000, 19.7 MB of data (Figure 9) is communicated to object storage at the start of each round. This inter-zone egress of data to storage across the entire protocol execution results in a total communication overhead of 1.18 TB. At cross-region egress pricing ($0.02/GB), this results in an increase of $23.64. Still, even with this increase, checkpointing remains cheaper than migration at our largest sample size tested. However, these results highlight that checkpointing benefits most from intra-zone deployment, where communication and storage transfer costs are effectively eliminated.

B Proofs

We prove that ThriftyMPC is secure against both a Dynamic MPC adversary (Definition 1) and a Checkpoint-Dynamic MPC adversary (Definition 2).

Our protocol execution is specified by an interactive Turing Machine referred to as $\mathcal{E}$, the environment. $\mathcal{E}$ represents everything external to the computation. The protocol progresses in rounds ℓ, where at the start of each round $\mathcal{E}$ generates and provides parties with inputs, receiving outputs from parties at the end of the round, and can interact arbitrarily with an adversary $\mathcal{A}$ throughout the duration of the round.

To prove the security of our protocol Π, we compare the real-world execution to the ideal world. We view the environment $\mathcal{E}$ as a distinguisher, interacting with our adversary $\mathcal{A}$ in the real world and with our simulated adversary Sim in the ideal world. If no environment $\mathcal{E}$ can distinguish between the real and ideal executions, then the two executions are perfectly indistinguishable. Thus, security comes from the fact that the adversary learns no more information than what could be learned in an ideal setting under a trusted functionality computing f.

In the real-world execution, the environment $\mathcal{E}$ provides inputs to the parties and interacts with the adversary $\mathcal{A}$. During the execution, $\mathcal{A}$ may dynamically corrupt parties. When a party is corrupted, the adversary learns that party's internal state and may send messages on its behalf for the remainder of the protocol. We assume that in every round ℓ the adversary corrupts at most a minority of servers. The protocol proceeds round-by-round in the presence of $\mathcal{A}$ and $\mathcal{E}$ until all parties produce outputs.

In the **ideal-world** execution of our protocol, the functionality f is computed by a trusted party. The participants send their inputs to the trusted party who computes f and passes the outputs back to the participants. Whenever the adversary corrupts a party, the simulator is notified of the corruption and must generate a simulated internal state and transcript for that party that is consistent with the protocol execution. Sim learns the corrupted parties inputs from the adversary and is given the outputs from the trusted functionality.

Sim interacts with the environment in the same manner as the adversary in the real world. The trusted functionality computes f and sends $\{Y_i\}_{i\in\mathcal{T}}$ to Sim, where $\mathcal{T}$ denotes the set of corrupted parties. The outputs of both the honest parties and the corrupted parties are the values returned by the trusted functionality.

B.1 Migration

We first provide the definition of security under migration in Definition 3.

Definition 3 (Dynamic MPC-Secure)**.** Let $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$ be a functionality, and let π be a ThriftyMPC protocol for computing f.

We say that π is secure against a Dynamic MPC adversary if for every adversary $\mathcal{A}$, there exists a simulator `Sim` such that for every vector of inputs $\vec{x}$:

$$\left(\text{view}_{\mathcal{A}}^{\pi}(\vec{x}),\ \texttt{output}^{\pi}(\vec{x})\right) \equiv \left(\texttt{Sim}(\vec{x}_T, f(\vec{x})),\ f(\vec{x})\right)$$

where T denotes the set of parties corrupted by $\mathcal{A}$ during the execution. The adversary $\mathcal{A}$ may adaptively corrupt parties across rounds in a non-fixed manner. Upon corruption of a party i, $\mathcal{A}$ learns its internal state, including its randomness, and message transcript up to the point of corruption. The simulator $\texttt{Sim}$ is given the inputs of corrupted parties at the time they are corrupted.

The view of the adversary $\mathcal{A}$ during an execution of π is defined as:

$$\text{view}_{\mathcal{A}}^{\pi} = \{\text{state}_i : i \in T\},$$

where state_i denotes the internal state of party i at the time of corruption, including its randomness, and message transcript up to that point.

We can show migration is secure against a Dynamic-MPC adversary by proving Theorem 1.

Theorem 1. *Let $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$ be a functionality, and let Π_{mig} be a ThriftyMPC protocol leveraging migration computing f as a layered circuit with N servers, and E rounds. Then, Π_{mig} is* Dynamic MPC-Secure *against any Dynamic MPC adversary $\mathcal{A}$ corrupting at most a minority of servers in each round ℓ*

Proof. We construct a simulator Sim that produces a view indistinguishable from the view of the adversary in the real world. First, note that our coordinator can be modeled as a synchronization primitive; this node has no knowledge of the data being computed, but simply signals for parties to proceed to subsequent rounds. The coordinator does not affect the information available to the adversary.

ThriftyMPC evaluates the functionality f as a layered circuit. In this execution model, the inputs to layer ℓ consist of the outputs of layer $\ell - 1$, represented as secret shares held by the servers. Thus, all messages exchanged between servers consist only of shares of intermediate circuit values. When using our migration preemption tolerance mechanism, ThriftyMPC leverages GRR's degree-reduction optimization [GRR98], which ensures that each round produces a fresh t-out-of-n sharing generated with independent randomness via GRR. This holds even when a compute party in any particular round is a substitute; no stale state is present because substitutes are deleted and replaced after a new spot instance is available. Consequently, for any dynamic MPC adversary that corrupts fewer than t parties in a round ℓ, the shares it obtains are information-theoretically indistinguishable from uniformly random shares. Furthermore, because GRR's degree reduction uses fresh randomness in each round, the adversary's observations across rounds are consistent with fresh valid t-out-of-n sharings of the evolving wire values. This holds regardless of which rounds our adversary chooses to corrupt parties and which parties they choose to corrupt, as long as thresholds are not exceeded.

Sim operates as follows: for every round ℓ in which the adversary corrupts a set of parties T^{ℓ}, Sim can generate uniformly random shares consistent with valid t-out-of-n sharings for that round and assign them as the internal state for the corrupted parties. Additionally, Sim can generate randomness and messages consistent with the protocol execution, which depend solely on these shares. Because GRR ensures that each round produces a correctly distributed fresh sharing, the simulator's generated view is information-theoretically indistinguishable from the real execution. Finally, since the protocol operates in the semi-honest setting, the output $f(\vec{x})$ observed by the adversary are identical in the two executions. Therefore, the joint distribution of the adversary's view in the real and simulated execution are indistinguishable.

□

B.2 Checkpointing

We define security under checkpointing in Definition 4.

Definition 4 (Checkpoint-Dynamic MPC-Secure)**.** Let $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$ be a functionality, and let π be a ThriftyMPC protocol for computing f.

We say that π is secure against a Checkpoint-Dynamic MPC adversary $\mathcal{A}$ if there exists a simulator `Sim` such that for every vector of inputs $\vec{x}$:

$$\left(\text{view}^{\pi}_{\mathcal{A}}(\vec{x}),\ \texttt{output}^{\pi}(\vec{x})\right) \equiv \left(\texttt{Sim}(\vec{x}_T, f(\vec{x})),\ f(\vec{x})\right)$$

where T denotes the set of parties corrupted by $\mathcal{A}$ during the execution. The adversary $\mathcal{A}$ may adaptively corrupt parties across rounds in a fixed manner. Upon corruption of a party i, $\mathcal{A}$ learns its internal state, including its randomness, and message transcript up to the point of corruption. The simulator `Sim` is given the inputs of corrupted parties at the time they are corrupted.

The view of the adversary $\mathcal{A}$ during an execution of π is defined as:

$$\text{view}^{\pi}_{\mathcal{A}} = \{\text{state}_i : i \in T\},$$

where state_i denotes the internal state of party i at the time of corruption, including its randomness, and message transcript up to that point.

We can show checkpointing is secure against a Checkpoint-dynamic MPC adversary by proving Theorem 2.

Theorem 2. *Let $f : X_1 \times \cdots \times X_m \to Y_1 \times \cdots \times Y_m$ be a functionality, and let Π_{mig} be a ThriftyMPC protocol leveraging checkpointing computing f as a layered circuit with N servers, and E rounds. Then, Π_{check} is* Checkpoint-Dynamic MPC-secure *against any Checkpoint-Dynamic MPC adversary $\mathcal{A}$ corrupting at most a minority of servers in each round ℓ*

Proof. This proof follows the same structure as the proof of security under migration. A key difference is that rather than our adversary being allowed to arbitrarily corrupt servers across rounds, corruptions are fixed. In other words, after the initial corruption threshold is reached, the adversary can only corrupt replacements of previously corrupted spot instances.

Checkpointing persists state to access-controlled storage buckets. The credential to any specific checkpoint data store is known only by its assigned server – not even the coordinator knows this credential. Therefore, when a server is replaced, the replacement server retrieves the same checkpointed state, but no additional information about other servers' shares becomes available. Because the adversary can corrupt at most t servers in each round, it can never obtain more than t shares of any secret. Consequently, for any Checkpoint-Dynamic MPC adversary that corrupts fewer than t parties in a round ℓ, the shares it obtains are information-theoretically indistinguishable from uniformly random shares. Furthermore, since the checkpointed state an adversary can obtain after initial corruptions corresponds only to shares of previously corrupted servers, the adversary cannot accumulate additional shares across rounds that would allow reconstruction of the secret.

Sim therefore operates similarly as in the migration setting, generating for each round shares consistent with valid t-out-of-n secret sharings for the corrupted parties, as well as randomness and messages. Because corruptions are fixed, the adversary cannot exceed the corruption threshold. Thus, at any point it obtains fewer than t shares of any secret, which are information-theoretically independent of the underlying value. Consequently, the joint distribution of the adversary's view in the real and ideal executions is indistinguishable. □