Cloudless and Mixclaves

Vikranth Srivatsa

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-184
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-184.html

May 19, 2023

Acknowledgement

Thank you to everyone who has supported me in my educational journey. I would like to especially thank my family and my friends for their support.

The work presented was a joint effort with students Alan Pham, Chris Douglas, and Mark Theis. I'd like to thank my advisors from my various projects: Moustafa Abdelbaky, Joseph E. Gonzalez, John Kubiatowicz, Scott Shenker, Yaodong Yu, and Yaoqing Yang.

Cloudless and Mixclaves

by

Vikranth Srivatsa

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph E. Gonzalez, Chair
Professor John D. Kubiatowicz

Spring 2023

Cloudless and Mixclaves

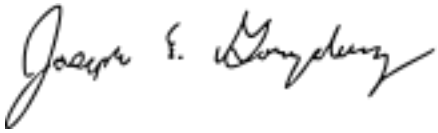
By Vikranth Srivatsa

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

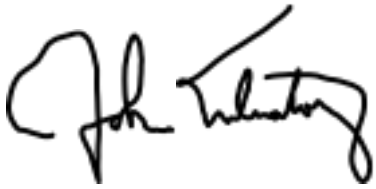


Professor Joseph E. Gonzalez
Research Advisor

5/12/2023

(Date)

* * * * * * *



Professor John D. Kubiatowicz
Second Reader

5/19/2023

(Date)

Abstract

Cloudless and Mixclaves

by

Vikranth Srivatsa

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Joseph E. Gonzalez, Chair

This thesis brings together two reports that focus on building a simplified secure compute abstraction across the cloud-edge. Cloudless is a serverless execution hierarchy that spans a multi-cloud to edge continuum and provides transparent function invocation across hybrid infrastructure. Cloudless provides a multi-cloud edge abstraction, enabling a simplified vendor-agnostic serverless computing model. Communicating with multiple regions brings about privacy concerns around authentication, anonymization, and integrity. Mixclaves is a metadata private messaging architecture that builds on hardware enclaves to provide a cost-efficient, low latency messaging service implementation deployable in public clouds. The work Cloudless and Mixclaves works towards the vision of an anonymous, cost-efficient, low latency, scalable computing paradigm that operates on multi-cloud and edge.

Contents

Acknowledgments

Thank you to everyone who has supported me in my educational journey. I would like to especially thank my family and my friends for their support.

The work presented was a joint effort with students Alan Pham, Chris Douglas, and Mark Theis. I'd like to thank my advisors from my various projects: Moustafa Abdelbaky, Joseph E. Gonzalez, John Kubiatowicz, Scott Shenker, Yaodong Yu, and Yaoqing Yang.

Chapter 1

Cloudless Computing: Serverless Across Hybrid Multi-Cloud/Edge Infrastructure

We introduce *Cloudless Computing*, a serverless execution hierarchy that spans a multi-cloud-to-edge continuum and provides transparent function invocation across hybrid infrastructure. It provides the illusion of infinite capacity at the edge, achieved by a simple RPC layer over the abstractions offered by serverless providers. The edge becomes an entry point, either serving function requests locally or routing them to the middle or the cloud, based on scheduling decisions and policies. We present a conceptual architecture and an open-source implementation of this abstraction and evaluate it using machine learning workloads. Our results show 7.3-29.3% improvement in response time and 2.9-73.9% reduction in cost compared to using a single cloud provider for some workloads.

Cloudless Computing: Serverless Across Hybrid Multi-Cloud/Edge Infrastructure

Anonymous Author(s)

Abstract

We introduce *Cloudless Computing*, a serverless execution hierarchy that spans a multi-cloud-to-edge continuum and provides transparent function invocation across hybrid infrastructure. It provides the illusion of infinite capacity at the edge, achieved by a simple RPC layer over the abstractions offered by serverless providers. The edge becomes an entry point, either serving function requests locally or routing them to the middle or the cloud, based on scheduling decisions and policies. We present a conceptual architecture and an open-source implementation of this abstraction and evaluate it using machine learning workloads. Our results show 7.3-29.3% improvement in response time and 2.9-73.9% reduction in cost compared to using a single cloud provider for some workloads.

1 Introduction

Computing has swung between centralization and decentralization for decades. Recently, we see a trend back towards decentralization, i.e., *away* from the cloud, driven by the requirements of real-time applications: low latency, intermittent connectivity, limited network bandwidth, privacy, and related regulations [1]. However, wide adoption of edge paradigms is hindered by the complexity of resource management, heterogeneity of infrastructure, and limited capacity [2].

Similarly, there is growing interest in adopting multi-cloud strategies for enterprise applications to avoid vendor lock-in and single provider failures, improve service reliability, adhere to changing business constraints and regulations, and reduce costs by taking advantage of different price offerings [3]. But due to similar infrastructure management complexity and heterogeneity reasons, multi-cloud strategies remain largely unmaterialized and are currently limited to running different applications on different clouds as opposed to running the same application across clouds [4].

The authors in [4] advocate for reciprocal peering as a key enabling step in achieving the Sky Computing [3,4] paradigm, where true utility computing is achieved by combining multi-cloud and edge computing via brokers that allow users to select the desirable cloud or edge provider.

In this paper, we demonstrate the viability of sky computing by proposing a serverless broker, which simplifies combining resources across multiple clouds and along the tiers of the cloud-to-edge continuum to support latency sensitive workloads while optimizing the placement of compute and data independently.

In particular, we introduce *Cloudless Computing*: a multi-tier serverless execution hierarchy that enables transparent function invocation across the tiers of the multi-cloud-to-edge continuum and allows users to easily switch between serverless providers. We present a conceptual architecture and an open-source implementation for such a framework, which supports function deployment and invocation across multiple clouds, remote-edge, and local-edge providers.

The **cloudless framework** exploits differentiated compute and pricing scheme from multiple serverless providers to lower the cost and response time for serverless workloads. It supports executing a series of dependent functions (i.e. a DAG of stateful functions), with each function independently deployed as its own service (similar to AWS Step Functions), with state transitions enabled by a uniform storage abstraction.

Cloudless leverages two adaptive scheduling policies (heuristics and linear programming) and periodic benchmarking to continuously optimize function invocations using multiple QoS objectives (e.g., cost, response time), and can be easily extended to use any scheduling policy or objective. Cloudless takes advantage of variability in performance across time, provider, and resources for different workloads [5,6] to optimize the overall response time and cost.

Cloudless supports commercial Function as a Service (FaaS) platforms including services from AWS, Azure, and GCP which operate on either the Cloud or the Remote Edge tier. Further, Cloudless supports function execution on Kubernetes clusters via multiple open-source serverless platforms (OpenFaas, Kubeless, Fission).

1.1 Contributions

In this paper, we introduce the cloudless computing framework to abstract the hybrid multi-cloud/edge continuum. We present this abstraction in order to reduce vendor lock-in, minimize costs, and latency.
We present the following **contributions**:

- We introduce Cloudless open-source framework, which supports a multi-tier serverless execution hierarchy across a multi-cloud-to-edge continuum that enables transparent function invocation and deployment across tiers and allows users to easily switch between cloud service providers.
- We provide adaptable dynamic scheduling systems (via heuristics and linear programming) in order to optimize over invocations multiple objectives (e.g., cost, response time) using benchmarked data
- We provide memory and cost heuristics that finds the optimal cpu and memory configuration for each system based on benchmarks for cloud providers and kubernetes systems
- We provide evidence for how current single-cloud systems have suboptimal cost, latency, throughput, and fault tolerance over cloudless.

2 Background

2.1 The Computing Continuum

There is a growing body of work in computing models and frameworks that focuses on the ability to execute computation outside of a cloud environment, either on end devices themselves or somewhere in the middle, to reduce latency [7, 8] . This work has proposed a variety of terms including edge computing [2], fog computing [9], and Mobile Edge Computing (MEC) [10].

We do not aim to taxonomize the terms that others have proposed, as the distinctions are not important for our purposes. Instead, we restrict ourselves to four distinct tiers of resources (see Figure 1) that span the continuum from the cloud to the extreme edge. We define these tiers and list their respective properties below.

The extreme edge refers to end devices where data is generated and/or consumed (e.g., sensors and actuators, cameras). Their processing power is very limited, but it can be enhanced by custom hardware (e.g., FPGAs). The extreme edge is closest to data sources (lowest latency) but exhibits the most heterogeneity in terms of hardware and kernels.

The local edge refers to local resources that are always on and one network hop (usually wireless but possibly wired) from the extreme edge (e.g., cloudlets [11], gateways, micro clusters). They have higher capacity and computational power, but are still limited. Edge resources can take advantage of virtualization, containers, or a serverless platform to simplify resource management.

The remote edge refers to resources between the cloud and the local edge (e.g., CDNs, ISP local stations, network PoPs, cell tower base stations). Remote edge resources can take advantage of virtualization, containers, or a serverless platform to simplify resource management. Further, a few commercial offerings [12–14] provide "as a Service" abstractions, where users can take advantage of a pay-as-you-go model, on-demand access, and elasticity.

The cloud refers to large data centers accessible over the Internet. The cloud provides the illusion of infinite capacity, elasticity, state-of-the-art hardware, pay-as-you-go billing, and on-demand access. However, cloud resources are furthest from data sources and actuators, thereby introducing additional latency when processing requests.

These tiers form a computing hierarchy similar to the memory-to-disk hierarchy, which we use as inspiration to implement a serverless execution hierarchy across the continuum. Similarly, for storage, there exists a parallel for each of the computing tiers with similar capacity and latency trade-offs (i.e., with larger capacity but higher latency storage at the cloud). Further, within each tier there are multiple storage abstractions (e.g., databases or key-value stores) with different trade-offs in terms of cost, latency, and consistency, which is important to consider when optimizing the placement of stateful functions across the continuum.

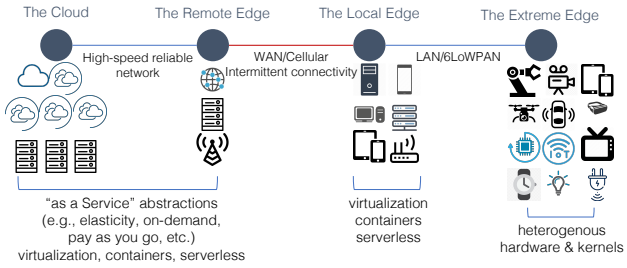


Figure 1: The four computing tiers in our system model that form a computing continuum.

2.2 Serverless Computing and the Continuum

Serverless computing has emerged as a paradigm for simplifying cloud programming [15, 16]. It provides: a) unified programming abstractions (e.g., event-driven function invocations), b) auto-scaling policies that can scale functions up or down from zero to many replicas based on workload, and c) a fine-grained, pay-as-you-go model, where users only pay for usage during function execution.

In this paper, we contend that serverless computing is a key enabler for programming the continuum and materializing sky computing. In particular, the serverless model decouples compute, network, and storage thereby allowing providers to

manage and scale them independently while simplifying process management and state migration [17, 18]. The cloudless framework takes advantage of this decoupling to optimize the selection of serverless providers as well as optimize the placement of function invocations and migration across providers.

3 Use Case Scenario & Systems Requirements

Global demand for air transportation is increasing with proliferation of air passengers and cargo transportation across rural and urban areas, involving Unmanned Aerial Vehicles (UAVs). Operating UAVs requires solving a diverse set of problems in real-time, such as handling video feeds, image recognition/detection, trajectory planning, and collision avoidance. The environment these UAVs are in may rapidly change as they fly through different geographical zones with differing levels of population. For instance, when traveling through a highly populated city, there are requirements for higher accuracy, thereby necessitating larger machine learning models.

Given the limited capacity of on-board computing, some of these computations have to be off-loaded to nearby cell tower base stations or all the way to the cloud (for non-latency sensitive workloads). Offloading computation from drones to edge, swarm, and for applications like deep learning has been explored in the past [19–21]. However, implementations of such approaches are hindered by the complexity of managing heterogeneous infrastructure across the continuum. This is further complicated by varying network bandwidth and latency as the vehicle travels from one area to another. As a result, while combining resources (on-board the vehicle and across the continuum) can provide the necessary scale and capacity for UAV workloads, dynamically allocating resources and distributing workloads under varying requirements and environments remains a challenge.

To support this use case and many other latency sensitive edge applications, we enumerate the following requirements:
Unified Framework across the Continuum: We need a framework that enables the transparent execution of workloads across distributed infrastructure. It should provide a uniform abstraction that allows developers to write code once and run it on heterogeneous infrastructure. The framework must support a wide variety of workloads with arbitrary code.
Multi-cloud and High Availability: During the duration of a UAV's flight, it might come across many different regions and zones with different network reliability. This requires a system that support a highly available multi-region/multi-cloud/multi-edge to provide optimal cost/performance over the flight time. It needs to be more available and more fault tolerant than any single provider.
High scalability and adaptability: The system must be highly scalable and adaptable to support varying workloads, dynamic network, compute, and other resource requirements, which might change over the course of the UAV's flight.
Edge Aware Scheduling: The local-edge (i.e., UAV on board computing) has limited compute, memory, and battery life. The system must transparently auto-scale up/down and in/out across the continuum to meet varying workloads, dynamic network, compute, and storage availability, while meeting local-edge capacity constraints, latency and cost requirements.
Extensible: the framework should be extendable to allow for new infrastructure providers, scheduling algorithms, and optimization objectives to be added over time.

4 Conceptual Architecture

Cloudless computing provides transparent function invocation across a serverless execution hierarchy that spans a multi-cloud-to-edge continuum. Following the use case requirements in Section 3, we present a conceptual architecture in Figure 2, which includes 1) one or more serverless platforms at each tier, 2) a control plane that monitors and schedules the execution of functions across platforms and tiers, and 3) a uniform storage abstraction. Note – In our work, we choose not to continue exploring the extreme edge because it requires custom hardware and kernel support.

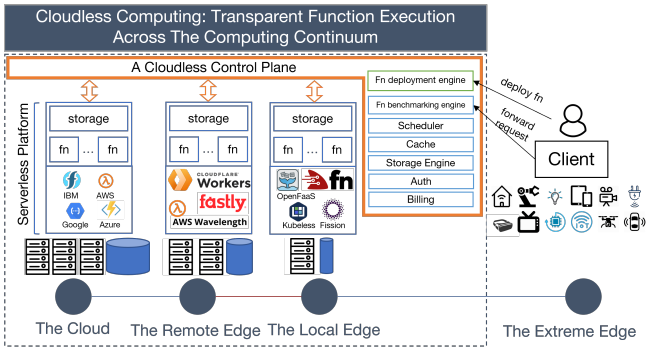


Figure 2: Conceptual architecture of a cloudless computing framework, composed of serverless platforms at each tier and a control plane that enables function invocations across tiers based on QoS objectives.

Cloudless Control Plane. The control plane is responsible for deploying new functions to tiers and invoking them based on QoS objectives and policies. It combines the components of an extensible framework, uniform execution, and scheduling. It includes the following components.

(a) *Function Deployment Engine*: prepares a function and deploys it to the different serverless platforms. Once deployed, the engine exposes an API endpoint, typically one per function per tier per platform, to invoke the function.

(b) *Function Benchmarking and Invocation Engine*: benchmarks and invokes workflows across tiers, including single functions and DAGs. The engine collects performance metrics for deployed functions based on requests

generated by a *client* at the extreme edge. The engine can also monitor local-edge utilization and function request rate to account for the limited capacity at the edge.

(c) *Universal Storage*: provides an interface that allows functions to store intermediate or final results. Underneath this layer, different storage providers at different tiers can be used, each with their associated cost and performance trade-offs. A *Storage Engine* monitors the performance (latency and bandwidth) and costs of data transfers across tiers and informs the Function Scheduling Engine.

(d) *Function Scheduling Engine*: schedules functions invocations across platforms and tiers. Engine decides where to run a function (or a DAG) based on collected metrics and edge utilization. The engine leverages a *scheduler* and different QoS policies (e.g., cost, response time) to forward requests to one or more platforms or tiers. The scheduling engine is extensible to new policies and adapts to new metrics.

(e) *Cache*: stores function information to optimize similar invocations. For functions that can be memoized, this will eliminate repeated calls.

(f) *Authentication & Authorization Service*: manages user credentials and security related tasks across providers.

(g) *Billing Engine*: administers and monitors billing and costs across all providers . It can monitor providers cost and inform the Function Scheduling Engine.

Serverless Platforms. Each tier contains one or more serverless platforms that support the deployment of new functions and the invocation of existing ones. These can be based on commercial offerings (e.g., AWS Lambda, Google Cloud Functions) or open-source and research platforms (e.g., OpenFaaS [22], Fission [23], OpenLambda [24], Cloudburst [18]). Cloudless extends serverless's high scalability properties. Typically, a FaaS stack is deployed on top of a cluster of resources and is composed of the following elements.

(a) *Resource Manager*: responsible for managing the underlying computing cluster. It provides mechanisms for monitoring nodes, handling failures, and scaling nodes.

(b) *Execution Environment*: encapsulates functions and its dependencies, used to deploy functions on heterogeneous infrastructure, and provide function isolation.

(c) *Scheduler*: responsible for orchestrating functions, monitoring their performance and availability, and autoscaling functions based on different metrics.

(d) *Front End*: responsible for deploying new functions, providing ingress (e.g., HTTP(s) endpoints) to route requests and invoke functions.

5 Implementation

We followed the conceptual architecture in Section 4 to develop a prototype implementation of the Cloudless framework. We describe the main components of the implementation below.

5.1 Control Plane

5.1.1 Function Deployment Engine

To support deploying functions across the continuum, we use Docker containers to simplify handling heterogeneous architectures. For providers, this removes function sizes limits (up to 10 GB), supporting various workloads. For open-source frameworks and the edge, we leverage Kubernetes for simplified infrastructure, auto-scaling, and network management.

The Function Deployment Engine can deploy a single function to multiple regions/clouds to allow for fault tolerance, larger throughput, and better availability. This comes at the cost of maintaining deployments and endpoints at each location, mostly comprising of low storage costs. The deployment engine has quality of life improvements to deploy in parallel and retry failed deploys.

5.1.2 Function Benchmarking and Invocation Engine

Compute Metrics. Developers can record custom statistics and information about their invocations. Custom benchmarking algorithms must inherit from a base class and implement two methods: a function that is run before invocation, and another that is run afterwards. Our current benchmarking functions includes the following information:

(a) *execution time*: Time to execute function (T_e).

(b) *network delay*: Time spent in the network (T_n) (including time to send a request and time to receive a response)

(c) *warm start*: Platforms that support caching can amortize function cold start costs across invocations. Routing and scheduling mechanisms within a platform can introduce additional latency or bottlenecks. Auto-scaling policies and the unit of scaling (e.g., processes, containers, VMs) can also impact performance. We chose to focus our analysis on invoking with warm containers. We record warm start state by checking if global cache was modified.

(d) *memory usage over time*: We poll memory usage every 0.1 seconds. This helps us estimate cost for providers where RAM usage is part of pricing (ex. Lambda charges more for compute tiers with higher memory limits).

(e) *Cloudless overhead time*: Control plane scheduling decision overhead (T_s)

Compute Cost We collect compute cost information, which is essential for the optimization of function placement. Each provider offers multiple levels of compute and memory configurations (i.e., compute bins). For example, AWS Lambda offers 13 compute bins with different CPU and

memory allocations, and different pricing. To simplify our optimization and determine cost information, we use the following cost models. Note: We exclude the free tier from our cost model. We consider the case for three different types of providers:

(a) *Commercial serverless providers*: services like AWS Lambda, Azure Cloud Functions, and GCP Cloud Functions or Cloud Run all offer pricing models that charge by execution time and/or memory. As a result, given execution time and memory usage of a function, we can estimate pricing per invoke for a function and workload.

(b) *Kubernetes providers*: it is difficult to estimate the cost per function for serverless offering where we spin up Kubernetes clusters and use open-source serverless platforms. We model cost for a single invocation as its execution time in seconds multiplied by the cost of the cluster per second.

(c) *Edge providers*: For edge providers, we use a simplifying assumption that running on the edge is free, since users will usually own the infrastructure at the edge.

Although the cost models above are simple, our framework is built such that it is easily extendable to different cost models (which may need different types of benchmarking information), and types of compute/memory nodes for each provider. Additionally, we have implemented a uniform mechanism to collect benchmarking information in a vendor-agonstic manner to minimize vendor lock-in.

Networking For each function, we look at the round trip delay for sending a request and receiving a response. We use an empty function packaged with all the same dependencies to quickly find the scheduling delay. To simulate a similar network delay to the function, we send an input packet and return output packets that is around the same size as the one used in the function. To support this, for each function, we also measure the average bytes of the input/output.

Using the dependency tree for multi function workflows, we measure the networking delays between, from, and to each of these functions. For these DAGs, we consider three different types of node networking. For example in Figure 4, there is the network delay between client and function 1, function 1 and function 2, and function 2 and client.

Storage Cost All three storage providers that Cloudless currently supports offer monthly pricing with costs associated for networking and storage. To approximate pricing, we consider a simple cost model:

c_m = cost/gb/month | s_{avg} = storage avg./month
c_{op} = cost/operation | m_u = months
e_{op} = expected operations

$$\text{storage cost} = c_m * s_{avg} * m_u + e_{op} * c_{op}$$

We significantly reduce storage cost by storing our intermediary DAG results temporarily– these results can be removed from storage after they are read by all functions nodes that need them. Similar to compute, we consider edge storage free.

5.1.3 Universal Storage

Cloudless enables stateful function execution and function DAGs by providing a thin abstraction layer on top of existing KV storage offerings. In the cloud, Cloudless uses AWS S3, Google Cloud Storage, and Azure Blob Storage to store the intermediate output of functions or persist the final results. At the remote and local edge, we leverage Redis as the underlying KV store. We create these resources in different cloud providers and use lithops storage [25] as a way to access and perform uniform GET and PUT operations. The different storage offerings have vastly different access patterns and requirements that change based on the workload [26].

5.1.4 Function Scheduling Engine

Cloudless maximizes QoS objectives by optimizing function configuration, invocation, and placement. Each provider has different function configuration (defined by the amount of memory and/or CPU available), with corresponding cost. The performance and cost of these providers can drastically vary over time [6]. Cloudless takes advantage of this variability to optimize function configurations and invocations.

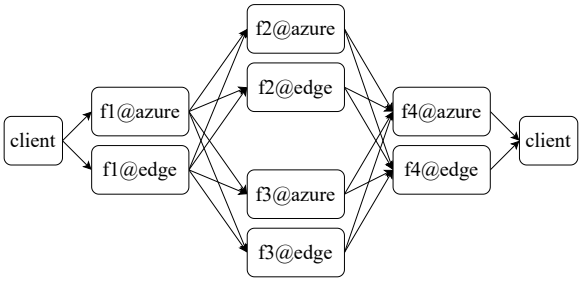


Figure 3: Search space of function placements with the providers azure and edge. The dependencies between the function are shown in Figure 7.

Given a list of providers, we create a search space of function placements (see Fig. 3). The graph edges encode storage cost and time (including read/write time) to move data between different providers. The graph nodes encode information about: provider location, function cost, and function response time. To reduce our search spaces, we use a heuristic to determine the best storage to read and write to based on the average read, write, and cost metrics for every storage. There are other formulations of the cloud-edge search space. Costless [27] presents an alternate approaches to pruning the search space by only moving from the edge to the cloud and considering combining nodes for better cost.

Cloudless implements two sample scheduling policies: periodic and live scheduling. Cloudless can be easily extended to include more policies.

Cloudless implements periodic scheduling via periodically running scheduling algorithms that are heuristics and linear

programming based. The linear programming policy optimizes the search space using a modified constrained shortest path algorithm. Since an optimal solution is intractable, we choose to minimize the overall cost sum of all edges and nodes. The heuristic scheduling based decision tree chooses the best function independent of every other function.

Cloudless also supports a live scheduling policy: it implements an invocation server that dynamically determines which provider to forward incoming requests to. Cloudless employs a multi-threaded worker architecture which maintains a shared queue with a buffer of jobs. The server stores recent latencies for each provider via a thread-safe queue, and the outstanding invocations via atomic operations. For live scheduling, we implement four different algorithms for selecting the provider to invoke: (1) least outstanding invocations (LOI) (2) least cost (3) least latency, where we select the provider that with the lowest moving average latency, and (4) linear combination of any subset of the above policies.

To make our approach more robust, we incorporate a probability of randomly selecting a provider for invocation, regardless of recent performance. This mitigates the possibility of overlooking providers which recovers from temporary performance degradation.

Live scheduling (1) effectively extends the Cloudless concurrency limit, (2) adapts to changing response times or costs, and (3) is fault tolerant against subsets of providers failures.

For scheduling based on cost, the scheduler assumes the edge is free and we leave modelling of the edge's cost model to other work. The scheduler will not schedule everything on the edge due to computer and memory constraints provided on the edge.

Workflows DAGs. Function orchestration of DAG workflows can simplify coordination of subsystems, and deploy complex workflows to multiple containers. This allows us to allocate the most appropriate resources for each subfunction to improve end to end response time, reduce cost, and improve security. Cloud providers like AWS, GCP, and Azure have workflow as a service offerings (AWS Step functions, Google Cloud Functions, and Azure Durable Function). However, none of the mentioned services support multi-cloud or combining cloud and edge resources. Serverless workflow frameworks also have limitations on the number of transitions, request size, and maximum number of workflow executions. Cloudless surpasses these limits by combining resources uniformly. For example, AWS has a 1MB limit on request sizes, while Cloudless request size is limited to as much storage provider can handle (e.g. 5TB with AWS S3).

We represent function workflows as a DAGs. Each node is a function and each edge is a state transition. We store a DAG representation of a function workflow at the control plane, including indications on where to deploy and invoke each function. We support parallel workflows by using a Thread Pool to invoke the node as soon as their parents execute.

To move data between two nodes or two functions, we use our storage abstraction to store the intermediate results. Figure 4 shows an example of moving data between two clouds. The best storage location to write to and read from between the nodes is calculated based on the benchmarks metrics. Further, we support writing to intermediate results to multiple locations allowing DAGs that fail midway to rerun without restarting the entire workflow.

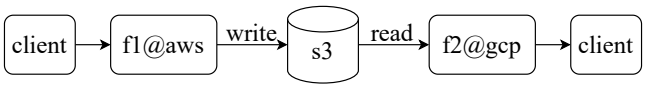


Figure 4: ex. of inter-cloud data movement. Function 1 in AWS writes output to S3. Function 2 in GCP reads from S3.

To support credentials for DAGs across the multi-cloud, we currently store the credentials required to connect to each of the different locations. However, in the future, we plan to implement a token-authentication service at the control plane using temporary IAM or JWT tokens.

5.2 Serverless Platforms

Cloudless supports arbitrary code execution using serverless providers in all tiers of the continuum.

In the cloud, Cloudless deploys and invokes functions using Google Cloud Run, AWS Lambda, and Azure Functions. It also deploys functions to open-source serverless frameworks including OpenFaaS [22] and Fission [23]), running on managed Kubernetes [28] clusters. Cloudless automatically creates these clusters using AKS, EKS, and GKE services from Azure, AWS, and Google respectively. To more closely simulate a serverless model, where cost is zero when no functions are invoked, we scale down pods and resources when not in use. By running Kubernetes clusters and provisioning our own compute instances, Cloudless is able to take advantage of special types of instances that are not provided by commercial serverless offerings. For example, Cloudless can select memory-optimized or high performance compute instances for a given workflow.

In the remote edge, Cloudless can deploy functions to a 5G cell tower base station offered by AWS Wavelength. It automatically creates a Kubernetes cluster and runs the open-source serverless frameworks on it.

At the local edge, Cloudless leverages a local Kubernetes cluster to run the open-source serverless frameworks. This allows us to maintain a serverless abstraction at the local edge where there are no dedicated serverless offerings.

5.3 General Pipeline

For Cloudless to be adaptive in changing environments, business requirements, and objectives, a general pipeline exists to optimize the deployment engine and the invocation engine.

(a) **Deployment**: Cloudless users begin by writing a config file with information like the filepath to their code and what providers they want to deploy to. Users then deploy

their function via Cloudless CLI to signal the Function Development Engine to begin setup locally: deployment state are created and stored inside a build folder. Then, providers from the config file are deployed. An endpoint is specified for each provider-function pair.

(b) **Benchmarking**: Cloudless collects metrics on compute, memory costs and other analytics of each provider. This is done either (1) periodically, including first deployment, and/or (2) live, as invocations occur naturally.

(c) **Scheduling**: We have two scheduling policies: (1) Periodic scheduling algorithms process periodic benchmarks to determine where to initially deploy functions to optimize for user defined objectives (i.e. cost, execution time, or a linear combination of both). If the scheduling determines it, we redeploy services required. (2) Live scheduling chooses where to dispatch invocations based on live benchmarking (recent peformance metrics) based on scheduling subpolicies including: least outstanding invocations (LOI), least cost, and least latency.

(d) **Cleanup Engine**: In order to minimize costs, the metrics from the scheduling algorithm can be used to clean up services that are not scheduled as optimal. Redundancy can be introduced by keeping multiple services deployed, and is dependent on business constraints on cost and uptime reliability. For serverless providers, costs are kept to a minimum since pricing is based on a pay-as-you go model. Costs incurred here are mainly storage.

The pipeline enables the UAV use case described in Section 3. It accounts for varying latency and cost for different workloads and adapts accordingly. The pipeline also helps users avoid vendor lock in. As benchmarking numbers like cost or execution times change, users using Cloudless can automatically switch between vendors to optimize for their QoS, and allow for function independence.

Finally, the Cloudless control plane can run at the local-edge (i.e., on-board computing of a UAV) or in the cloud. When running at the local-edge, invocation requests can run locally or be routed to the remote-edge or different clouds based on the control plane decision and local-edge utilization. Alternatively, the control plane can run in the cloud to support multiple workloads (i.e., from multiple UAVs). Incoming requests can be routed based on a global optimization, which can account for the flight paths of multiple UAVs.

6 Evaluation

As discussed in Section 3, we motivate the need for a multi-tier network continuum via UAVs. UAVs have flight patterns that may cause them to change the closest cloud regions they are in and have big shifts in workloads. While UAVs are heavily constrained by battery-life, we leave the modeling of this and other constraints to other work, and chose to model the work to have limited edge compute and memory. For our experiment, we use a Intel(R) Xeon(R) CPU E7-8870 v3 @ 2.10GHz to represent the edge. We use the same local computing cluster for all experiments ran on the local-edge to keep results consistent.

The evaluation is split into the multiple sections to demonstrate the different parts of a UAV workload and to evaluate the Cloudless framework. Since UAVs have variable networks, we first evaluate network overheads, latencies, and bottlenecks. In order to assess the workload of a UAV moving to different locations, we evaluate our framework using different sizes [29] of the image classification model ResNet [30] to model changing requirements and workloads. To model complex periodic benchmarking scheduling as the UAV moves between regions, we use a DAG as represented in Figure 7, demonstrating that different workloads can be scheduled to places that best suite the workload (such as compute heavy jobs running in compute heavy machines). Since UAV flight patterns change over time, we measure variance and adaptability of a UAV workload over time, including analyzing: fault tolerance, live scheduling, and response time variance.

By providing these different modes of evaluation, we hope to demonstrate the adaptability of Cloudless rather than analyzing the benchmarking that occurs at this point in time.

6.1 Networking

Function location makes a large impact on the network performance over different benchmarks. Additionally, providers have different service availability depending on region. For our benchmarks, we chose to deploy functions and clusters to the nearest region to the control plane on the edge: (AWS: us-west-1, GCP: us-west2, AZURE: us-west, AWS Wavelength: us-west-2-wl1-sfo-wlz-1).

Serverless systems have a scheduling delay, of finding a warm container or spinning up a new one, to route the invocation to a container. Docker runtimes generally require a higher scheduling delay than managed runtimes provided by cloud serverless providers (such as the Python runtime); however, Docker runtimes allow for higher limits on deployment filesize and timeouts. Packet size and input processing can contribute to scheduling delay. To measure scheduling delay, we deploy an empty function packed in the relevant Docker containers and benchmark data over 100 invokes. We verify there is no network bottleneck for parallel invocations by testing the response time to each of cloud with increasing payload size.

6.2 Function invocation

With the motivating example of an UAV moving to different locations requiring different accuracies, we evaluate our framework using the image classification model ResNet [30] (trained on ImageNet), at different ResNet sizes. Previous UAV work like [29] have also used the ResNet architecture as a representative drone workload. Additionally, ResNet is found in many serverless benchmarking workloads like Sebs [5]. The different ResNet sizes can be motivated by

the UAV moving to a highly populated location, requiring different accuracy threshold.

Previous works such as Sebs [5] describe the presence of the tradeoff between time and cost as different offerings withing serverless providers are chosen. We extend this work to also benchmark over node type since they heavily affect the performance of the workload. We replicate similar results with the performance-cost tradeoff. For the rest of our paper, we consider picking the tier that is the most cost effective.

We choose to benchmark on the ResNet sizes 18, 50, and 152, which have different compute and memory requirements [31]. The larger ResNets have higher accuracy. We ran a workload with 4MB payload of 30 images, and computed classifications serially (batch size of 1). Figure 5 shows that the fastest provider changes depending on the model: for example, AWS fission performs relatively better on ResNet18, but worse on higher ResNet sizes. These differences in QoS metrics between providers for different workloads necessitates the use of Cloudless framework to automatically pick the best provider and position to compute.

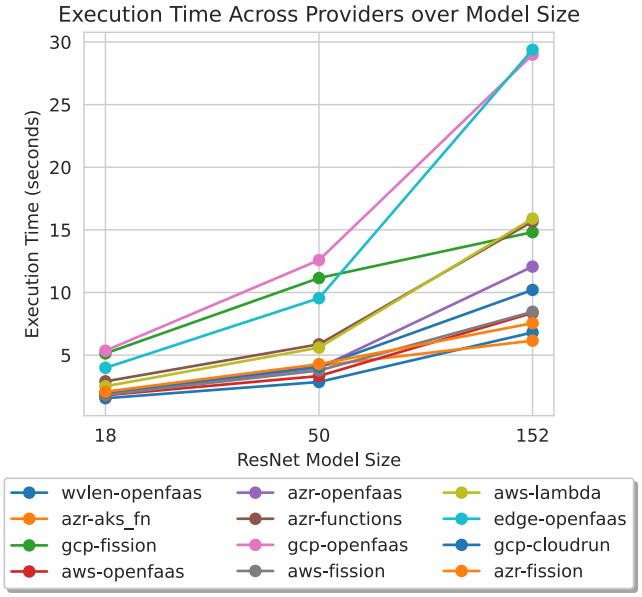


Figure 5: Execution times of ResNet over various providers over different model sizes. The differing ResNets have differnt memory and compute footprints, causing different providers to perform better than others.

6.3 Storage

UAVs can choose to persist results or temporary computation at different locations, which has various performance and cost tradeoffs. Cloudless supports the storage providers: AWS S3, GCP Cloud Storage, Azure Blob Storage, and Redis.

For each function, Cloudless runs a set of storage benchmarks to compute the amount of average read and write cost to every datastore and analyze patterns, including bottlenecks at large packet sizes. Figure 6 shows a heatmap plot of writing to the different locations with 64MB.

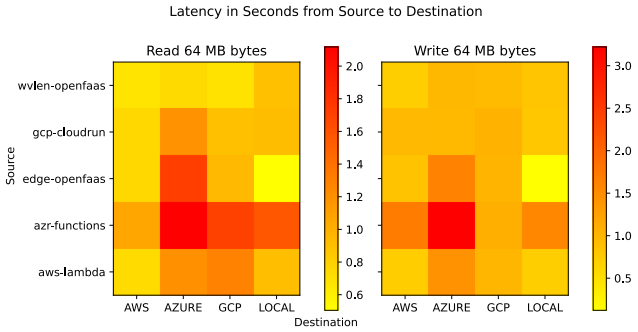


Figure 6: A heatmap of the network delay from source compute nodes to destination storage for a 64MB payload. We see differing patterns of read & write times that can be optimized.

We also ran a set of storage benchmark scans from 1B to 64MB to analyze for any interesting patterns between different payload sizes in Figure **??**.

6.4 DAGS and Periodic Scheduling

To extend our motivating example to more complex workflows, we use the DAG in Figure 7 to represent our workflow. We first take an input image, preprocess it to resize to the correct size, run both VGG and ResNet, and then persist the data to every single cloud storage. We decided to use VGG to represent the common task of object detection, while ResNet to represent the common task of classification. We used the preprocess function to represent the connection the client, which can represent a video processing node that constantly processes input images.

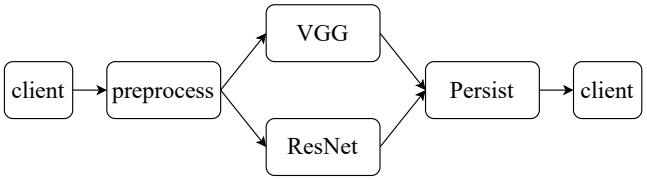


Figure 7: Sample UAV serverless workflow

Using the strategy to collect DAG metrics mentioned in Section 5.1.2, we benchmark the dag, computing the network delay, storage, and execution time. Depending on the objective required, our scheduler picks an optimized path through the searching the search space. Using the linear programming heuristic described in 5.1.4, we compute placements for time, cost, and few configurations on a single provider(such as all on AWS). From the results in Table 1, we can see that scheduling across multi-cloud performs better than any single provider or location. We can also see the benefit of combining edge and cloud resources. Based on the workload and the network connections with the client machine, the exact network

results and time will vary. Using periodic scheduling, we can periodically find the best location to run every function.

6.5 Evaluation over time

Since UAVs constantly are constantly running and move around during the flight, we ran an evaluation over time to measure changes in latency and cost. We perform a thousand consecutive invokes on the providers we are studying. We also ran a sample experiment of evaluation over a short number of multiple days as well, but did not notice a significant change. Our results show a noisy variation over time. This is due to the variability of serverless infrastructure or network, which can be another stability metric to schedule over. The graph also shows the edge used has higher variability(due to other processes running on the edge), similar to live UAV.

6.6 Live Scheduling

Cloudless supports a live scheduling policy for a UAV that needs realtime scheduling as it flies. We evaluated this policy by having Cloudless adapt to concurrency limits of each provider without awareness of what they are. We scaled the number of threads making invocations over time. We utilized Lambda with global concurrency limit of 10; Azure with a simulated limit of 20; the GCP service autoscaling limit of 30 and container concurrency limit of 1; and Edge with a simulated limit of 5. Note that settings deviate from defaults, which could impact performance. As shown in Fig. 8 Cloudless supports the concurrent requests increasing over time, past individual cloud limits, and reaches higher throughput than any individual provider. Between time 0 and 15, most of the invocations go to AWS because its the fastest. After second 30, all cloud providers are used as we go above the concurrency limit of GCP and Azure combined.

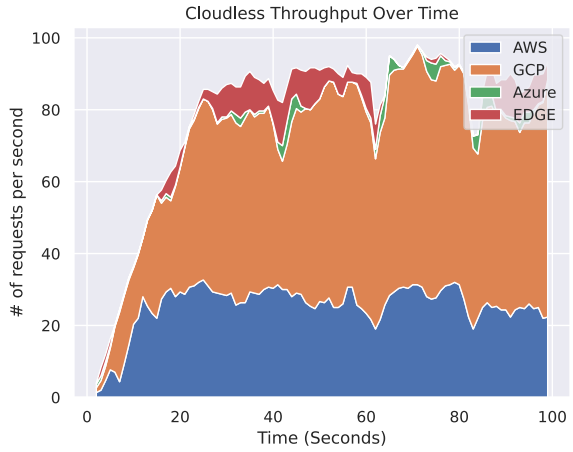


Figure 8: Throughput of Cloudless on each provider on ResNet18, smoothed with window size of 3. AWS has a concurrency limit of 10, Azure 20, GCP 30, and EDGE 5. Number of concurrent global requests are increased over time from 0 to 40 seconds.

Our results demonstrate that Cloudless does the following: (1) Cloudless addresses the problem of limited concurrency limits in individual providers via increasing total throughput by invoking to multiple providers. (2) Cloudless addresses issues with changing concurrency limits for functions by adaptively invoking to other providers when invocations go past the concurrency limit in one. For example AWS has a global account concurrency limit across all functions within an account. (3) Cloudless adapts to changing workloads and response times by minimizing cost and latency in real time. As demonstrated in section 6.5, response time can change significantly over time on edge, and workloads can change over time.

The live and periodic scheduling policies show the extensibility of the scheduling provided by Cloudless.

6.7 Fault Tolerance

To evaluate the fault tolerance of the system, we ran an experiment that began with multi-cloud servers and the edge, and turned off each of the clouds at different intervals until only the edge was available (Figure 9). The scheduler chooses the lowest latency cloud that is available, and we see that its able to instantly change this over time. Cloudless continues to run when clouds fail.

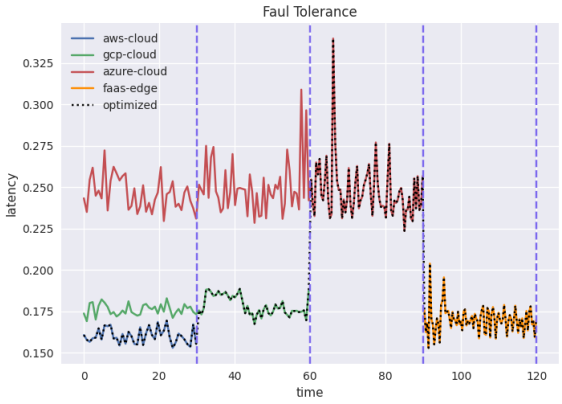


Figure 9: The graph displays the lowest latency policy. It sends all invocations to the fastest provider available at any given timepoint. After each vertical line, we simulate a cloud returning errors for all invocations(from AWS, GCP, then Azure). When no clouds are available, the edge is used.

7 Discussion

Argument for Multi-cloud Adoption of multi-cloud strategies in enterprise has been gaining attention in recent years; however, there is still a lack of widespread adoption of solutions that arbitrages many providers [4]. Some argue the lock-in of data stores and the complexity of current industry applications are deterrents for enterprises to adopt multi-cloud.

Table 1: We display the scheduling results of the example DAG workflow in Figure 7. The following is optimizing for time given an unlimited cost constraint.

name	time(s)	cost($)	persist	resnet	preprocess	vgg
default lp	2.531045	0.000298	aws-openfaas	aws-openfaas	edge-openfaas	azr-functions
no edge	2.977129	0.000298	aws-openfaas	aws-openfaas	aws-openfaas	aws-openfaas
aws only	2.866828	0.001145	aws-openfaas	aws-openfaas	aws-openfaas	aws-openfaas
gcp only	3.581388	0.000982	gcp-cloudrun	gcp-cloudrun	gcp-openfaas	gcp-openfaas
azure only	2.730473	0.000307	azr-fission	azr-fission	azr-fission	azr-functions

Our focus is not to make a statement about the economic considerations for the future of multi-cloud, but rather to develop a technological solution that can enable the use of multi-cloud.

In terms of developing Cloudless, differences in API made it challenging to find the best option (such as the autoscaling and caching) for each workload even with the serverless paradigm. Cloudless navigates around these limitations with periodic benchmarking, but future multi-cloud tools need to be careful to consider the subtle differences in the offerings.

Security, Authentication and Billing There is a lack of a single authority for billing scenarios and credentials, which leads to a wider attack surface and other security concerns. We hope that future multi-cloud solutions will work on providing better solutions to identity and billing management. State can also be shared across invocations for warm serverless invocations, which can potentially leak of customer data. Better isolated function caching might help mitigate attacks.

Storage While running our experiments, a key insight we found is: when moving between clouds, the indirect path to access storage can be faster to due to network congestion and potential egress limiting. However, for large amounts of data movement, egress fees can be expensive and act as a limiting factor. We hope future work can optimize these solutions.

Scheduling and Workload Shifts A generalized serverless system may struggle in making good assumptions about the nature of workloads in order to optimize itself, in part due to shifts in the distribution of data. Cloudless addresses this via its extensible scheduling system. For a DAG workflow with varying workloads on each node, we demonstrate the ability to optimize for different requirements. For example, secure computation can be run on an enclaves in the cloud while parallel jobs can run in compute heavy nodes. Furthermore, depending on the workload and the data requirements, the different network tiers can play a large part in the scheduling. We hope others will take advantage of this layered architecture to better localize data and compute.

Debugging and Logging The multi-cloud brings challenges in operation and maintenance. This is due to lack of debugging dashboards, lack of metrics, and longer deploy times. In the future, we hope to support a better logging system to nicely display the logs across the providers.

Adapatability of the Framework Lot of the results mentioned above are based one point in time for a specific configuration. An adaptable framework like Cloudless helps by optimizing for resources at different points in time. These numbers may dramatically change based on the region, bandwidth, latency and business constraints.

Kubernetes Deciding to provide serverless on Kubernetes comes with some tradeoffs. Kubernetes has open source serverless offerings that simplify building multi-cloud serverless. However, extreme edge kubernetes support is limited. Additionally, we found that Kubernetes clusters creation was much slower than serverless offerings from the vendors we support. There is also additional cost of operating the overhead across all providers but edge.

Fault Tolerance Cloudless develops fault tolerance via deploying to multiple providers, staying available as long as one of the providers remains online and one of the storage options remains accessible. Cloudless users are able to choose how redundant they want their compute and storage to be, with a trade off of higher cost for more availability. We plan to examine fault tolerance in depth in a future work.

7.1 Framework Details

Our framework is designed to extensible and flexible to use for any benchmark and any workload. Below we display how the DAG shown in 7 can be written in code. We designed the framework to be very similar to other serverless frameworks.

Listing 1: The 4 Functions used to represent the steps of the DAG. The input of each function is the previous step in the DAG. These functions can also acess the storage if needed

```
def resnet(preprocess): ...
def vgg(preprocess,): ...
def preprocess(event): ...
def persist(resnet, vgg): ...
```

In order to validate Cloudless's support of different types of functions, we implemented the PyBenchmarks and FunctionBench [32]. These will be available as open-source samples.

Developers can also import the cloudless library in python and use custom providers for all tasks including deployment and invocation. Custom providers must inherit from our base class and implement a number of abstract methods. We have 14 provider classes which developers can use as an example.

8 Related Work

Previous work discuss the multi-cloud and edge in context of performance, architecture, cost, storage, compute offloading, and workflows. However, while most of these works consider these factors independently, we use the Cloudless framework to examine the combination of these resources together.

Serverless Platforms Performance Evaluation. Prior work categorizes and taxonomizes existing serverless providers. The authors in [33] evaluate the performance of commercial serverless offerings and provide insights into architecture, resource utilization, and performance isolation efficiency for Amazon, Azure, and Google. Similarly, [5, 32] proposes a set of workloads for evaluating cloud serverless functions, which can be used to extend our workloads. [34] and [35] compare the features and performance of four open-source frameworks using a micro cluster in the cloud and at the edge respectively. The authors of [36] examines the throughput of different cloud providers and proposes a different architecture for the control plane.

Edge and Hybrid Serverless Frameworks. [37] presents a framework for low-latency offloading of computation via serverless at the edge. Unlike Cloudless, it does not include cloud resources or account for finite resources at the edge. [38] presents a vision and challenges for a framework where real-time analytics are processed on edge and advanced analytics on the cloud. Our work extends this vision to all tiers of the computing continuum. [39] proposes using a proxy at Edge to dynamically invoke serverless functions either locally via IoT devices, or cloud. The proxy uses historical execution times to determine routing. We found that embedded devices are too slow to support a serverless platform, and that offloading computations to an edge cluster was faster. [40] proposes a serverless edge platform tailored to AI applications, supporting serving, training, and monitoring AI models at the edge, while scheduling according to policies like privacy and regulations. The work Costless [27] discusses the combination of cloud and edge resources by creating a search space that moves function execution from the edge to the cloud. We build on top of this work to support resources in both cloud and edge platforms.

Serverless Frameworks: Lithops [25] supports running python programs on multiple serverless providers in massively parallel fashion. However, lithops doesn't support combining cloud edge resources, workflows, and open source severless platforms. We use lithops in order to create the storage abstraction across the mulit-cloud. Serverless framework [41] supports building and deploying functions to multiple cloud providers. We take inspiration to extend a similar invoking and deploying interface with workflows, storage, and more providers.

Storage: Other work has explored building a storage abstraction across the multi-cloud. DepSky [42] considers storing reliable and secure data. Anna [43] builds a distributed high performance key value storage, extending the limitations of current storage systems. We hope to extend this work to include a better storage abstractions, options, caching and optimizations for the multi-cloud.

Serverless Workflows: Another line of research is around analyzing and building function workflows. [44] studies the performance of AWS Step Functions, Google Cloud Composer, and Azure Durable Functions. The authors of SAND [45] build out a way to optimize performance of multiple function chaining and interaction. However, these works don't consider workflows across mult-cloud. With future work, we hope to be able to generate and compare the performance across these different workflow composers.

9 Conclusion

We presented Cloudless, an extensible generic framework that handles and schedules serverless execution over multiple tiers and adapts to changes in the environment. We defined a variety of providers, measure various benchmarking metrics over storage and compute, implement scheduling algorithms, and implement a number of memory and time optimizations. However, these serve as an initial set to evaluate our framework, and is easily extensible to improve. We show that in its current state, Cloudless is adaptable.

Using the Cloudless framework, users can dramatically cut costs, simplify stateful workflows, and remove vendor lock-in issues. Cloudless also shows the benefit of multi-cloud and how it's better than any single provider. As more multi-cloud frameworks like Cloudless are created, we speculate an overall trend toward cloud specialization based on the dramatic differences in cost, rel ability, and offerings. We hope that using the cloudless framework, we can move closer to a true "deploy once invoke everywhere."

References

[1] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[2] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[3] Katarzyna Keahey, Mauricio Tsugawa, Andrea Matsunaga, and Jose Fortes. Sky computing. *IEEE Internet Computing*, 13(5):43–51, 2009.

[4] Scott Shenker Ion Stoica. From cloud computing to sky computing. in proceedings of the workshop on hot topics in operating systems. pages 26–32.

[5] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A server-

less benchmark suite for function-as-a-service computing. *CoRR*, abs/2012.14132, 2020.

[6] Samuel Ginzburg and Michael J. Freedman. Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, page 43–48, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: concepts, applications and issues. In *Proceedings of the 2015 workshop on mobile big data*, pages 37–42, 2015.

[8] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018.

[9] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[10] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[11] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[12] Ben Rowdon. Wavelength, 2000.

[13] The edge cloud platform behind the best of the web.

[14] The cloudflare global network: Data center locations.

[15] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[16] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[17] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 10. ACM, 2013.

[18] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[19] Xiangwang Hou, Zhiyuan Ren, Wenchi Cheng, Chen Chen, and Hailin Zhang. Fog based computation offloading for swarm of drones, 2022.

[20] Bo Yang, Xuelin Cao, Chau Yuen, and Lijun Qian. Offloading optimization in edge computing for deep-learning-enabled target tracking by internet of uavs. *IEEE Internet of Things Journal*, 8(12):9878–9893, 2021.

[21] Wei Chong Ng, Wei Yang Bryan Lim, Zehui Xiong, Dusit Niyato, Chunyan Miao, Zhu Han, and Dong In Kim. Stochastic coded offloading scheme for unmanned aerial vehicle-assisted edge computing, 2022.

[22] OpenFaaS Ltd. Openfaas. https://www.openfaas.com/.

[23] Fission. https://fission.io/.

[24] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[25] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez. Outsourcing data processing jobs with lithops. *IEEE Transactions on Cloud Computing*, (01):1–1, nov 5555.

[26] Z. Daher and Hassan Hajjdiab. Cloud storage comparative analysis amazon simple storage vs. microsoft azure blob storage. *International Journal of Machine Learning and Computing*, 8:85–89, 02 2018.

[27] Tarek Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.

[28] Kubernetes. https://kubernetes.io/.

[29] Xiaoliang Wang, Peng Cheng, Xinchuan Liu, and Benedict Uzochukwu. Fast and accurate, convolutional neural network based approach for object detection from uav. In *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*, pages 3171–3175. IEEE, 2018.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[31] albanie. Memory consumption and flop count estimates for convnets. https://github.com/albanie/convnet-burden.

[32] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[33] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 133–146, 2018.

[34] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco, et al. An evaluation of open source serverless computing frameworks. In *CloudCom*, pages 115–120, 2018.

[35] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. An evaluation of open source serverless computing frameworks support at the edge. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642, pages 206–211. IEEE, 2019.

[36] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. On merits and viability of multi-cloud serverless. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 600–608, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Claudio Cicconetti, Marco Conti, and Andrea Passarella. Low-latency distributed computation offloading for pervasive environments. In *2019 IEEE International Conference on Pervasive Computing and Communications (PerCom*, pages 1–10. IEEE, 2019.

[38] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro Jakimovski, Sasko Ristov, and Radu Prodan. A serverless real-time data analytics platform for edge computing. *IEEE Internet Computing*, 21(4):64–71, 2017.

[39] Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–8. IEEE, 2018.

[40] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge {AI}. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[41] The serverless framework. https://serverless.com.

[42] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Trans. Storage*, 9(4), nov 2013.

[43] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Eliminating boundaries in cloud storage with anna. *arXiv preprint arXiv:1809.00089*, 2018.

[44] Jinfeng Wen and Yi Liu. An empirical study on serverless workflow service. *arXiv preprint arXiv:2101.03513*, 2021.

[45] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 923–935, USA, 2018. USENIX Association.

[46] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[47] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[48] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. Numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[49] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[50] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[52] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[53] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[54] Kyriakos Kritikos and Paweł Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168. IEEE, 2018.

[55] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[56] Moustafa AbdelBaky, Manish Parashar, Hyunjoo Kim, Kirk E Jordan, Vipin Sachdeva, James Sexton, Hani Jamjoom, Zon-Yin Shae, Gergina Pencheva, Reza Tavakoli, et al. Enabling high-performance computing as a service. *Computer*, 45(10):72–80, 2012.

[57] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, D Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[58] Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. Cost-aware cloud bursting for enterprise applications. *ACM Transactions on Internet Technology (TOIT)*, 13(3):1–24, 2014.

[59] Tian Guo, Upendra Sharma, Timothy Wood, Sambit Sahu, and Prashant Shenoy. Seagull: intelligent cloud bursting for enterprise applications. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 361–366, 2012.

[60] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards high-performance serverless computing. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.

[61] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 57–70, 2018.

[62] *HiPS '21: Proceedings of the 1st Workshop on High Performance Serverless Computing*, New York, NY, USA, 2021. Association for Computing Machinery.

Chapter 2

Mixclaves: Enclave-Based Mixnets

All secure messaging systems protect the content and integrity of users' messages, but the oblivious routing of messages concealing who communicates with whom (metadata-private messaging) is increasingly crucial for privacy. Existing techniques conceal routing metadata using *mix networks* (mixnets) made up of multiple nodes that batch and forward traffic to confound traffic analysis. State-of-the-art mix networks remain resilient to a passive global adversary even as attackers compromise up to 20% of the mix nodes.

As infrastructure moves to the cloud, threat models for metadata-private messaging must assume an adversary that is both active and even present on machines routing user data. This paper proposes *Mixclaves*, a scalable, metadata-private messaging architecture that builds on hardware enclaves to provide a cost-efficient, low-latency mixnet implementation deployable in public clouds. Building on stronger guarantees provided by enclaves not only simplifies the implementation of mixnets, it also admits novel features and lower operating costs. Compared to Loopix and Groove, two popular mixnet implementations, *mixclaves are 54% cheaper on cost to achieve the same message throughput.*

Mixclaves: Enclave-Based Mixnets

Mark Theis*
UC Berkeley

Chris Douglas*
UC Berkeley

Vikranth Srivatsa*
UC Berkeley

ABSTRACT

All secure messaging systems protect the content and integrity of users' messages, but the oblivious routing of messages concealing who communicates with whom (metadata-private messaging) is increasingly crucial for privacy. Existing techniques conceal routing metadata using *mix networks* (mixnets) made up of multiple nodes that batch and forward traffic to confound traffic analysis. State-of-the-art mix networks remain resilient to a passive global adversary even as attackers compromise up to 20% of the mix nodes.

As infrastructure moves to the cloud, threat models for metadata-private messaging must assume an adversary that is both active and even present on machines routing user data. This paper proposes *Mixclaves*, a scalable, metadata-private messaging architecture that builds on hardware enclaves to provide a cost-efficient, low-latency mixnet implementation deployable in public clouds. Building on stronger guarantees provided by enclaves not only simplifies the implementation of mixnets, it also admits novel features and lower operating costs. Compared to Loopix and Groove, two popular mixnet implementations, *mixclaves are 54% cheaper on cost to achieve the same message throughput.*

KEYWORDS

mixnets, hardware enclaves, metadata-private messaging

> We kill people based on metadata.
>
> Gen. Michael Hayden, former NSA Director, 2014 [23, 42]

> They are not looking at people's names, they're not looking at content, but by sifting through this so-called metadata, they may identify potential leads with respect to folks who might engage in terrorism.
>
> Barack Obama. 2014 [23]

1 INTRODUCTION

The scope of state-sponsored and corporate surveillance is so widespread, it is no longer covert; it is assumed. While communication providers safeguard users' confidentiality and integrity with authenticated encryption, communication *metadata* implicate users in friends' and acquaintances' lives. Churn in relationships and fluid boundaries of illicit conduct over time make it impossible to freely associate without fear of an unjust inference made using those metadata. As concrete examples, a logged call to a medical specialist could raise one's insurance premiums, without revealing any information about the content of the call. Communication metadata among company executives and their legal council reveals material, non-public information about a potential merger.

The series of leaked documents made available by Edward Snowden [26] revealed extensive, passive surveillance programs operated by the US Government. These programs primarily targeted metadata from communications, such as monitoring with whom and when someone communicates, rather than the content of communication. With the revelations that the capabilities of intelligence agencies approach that of passive global observers, leaving the metadata of communications exposed has been a dangerous attack vector.

Instant messaging has quickly become the dominant form of remote communication compared to traditional phone calls. The majority of these messages are sent over Short Message Service, an insecure service that is part of almost all mobile phones.

Secure and private messaging is a growing desire among the general public, especially for journalists, whistleblowers, activists, business executives, and those involved in the government or elections. The leading secure communication services Signal [37], WhatsApp [47], iMessage [28] support end-to-end encryption, guarding the contents of a communication, but they still expose metadata and do not have mechanisms to keep communication entirely anonymously.

Continued work has emerged to enhance privacy, known as metadata private messaging protocols. These aim to completely hide any information someone could learn about a user's communications. At the heart of these designs are mix networks, or mixnets. Mixnets are a series of proxy servers that bounce messages through the network. Similar to Tor, messages are wrapped in encrypted layers, with a given layer only readable to a designated node. Mixnets assume any node in a network may be untrusted. This affects the number of nodes a message must route through to achieve privacy.

A significant limitation of previous mixnets has been their high latency. For example, Vuvuzela [45] exhibits a 55-second end-to-end latency, while Pung [5] and Stadium [43] are even higher. Groove [9], one of the most recent designs that makes some of the strongest privacy guarantees, operates with 30-second or greater latencies at load. Such performance hinders the adoption of this approach to metadata-private messaging. One design in particular, Loopix [38], does make progress with latencies near one to two seconds. However, it makes weaker privacy claims compared to designs like Groove. We chose Loopix as our reference design, but aim to provide stronger privacy guarantees.

Recent work expanding the Internet architecture to support interposition has made general computing on secure hardware more accessible. Specifically, the Extensible Internet proposal [8] offers service attestation while running in secure enclaves on the public internet. With more widely available secure computing, we may form a stronger assumption about the hardware running the nodes of a mixnet.

*All authors contributed equally to this research.

CS262a Fall 2022, Advanced Topics in Computer Systems, UC Berkeley

In this project, we conjecture that enclaves simplify the assumptions for the number of nodes to run a probabilistically secure mixnet. We present **Mixclaves**, a design and implementation that utilizes enclaves to run a mixnet. Mixclaves withstand powerful adversaries that can observe all traffic flowing through a network and that may even control underlying hardware and a majority of other clients. We compare the performance of our approach running in enclaves to the traditional mixnet design Loopix operating on commodity hardware.

We claim the following contributions. To the best of our knowledge, this is the first implementation of a mixnet that relies on hardware enclaves to not only facilitate metadata-private communication asserting differential privacy, but also to enable dynamic scaling via a covert control channel.

Cost. Partitioning the mixnet into trust domains can collapse the mix network to a single node, saving up to 82% in cloud settings.

Elasticity. Metrics from enclave mix nodes are securely sampled and used to scale mixnet capacity within a trust domain by adding nodes to the mixnet.

Anonymous administration. Monitoring and control traffic are mixed with synthetic and real traffic from users.

Oblivious mix nodes. Memory traces of buffered traffic cannot be correlated within mix node enclaves. By leveraging the confidentiality and attestation guarantees of hardware enclaves, the Mixclaves architecture can provide more robust guarantees than a mix network. These guarantees apply to a single-node Mixclave.

The paper is structured as follows. Section 2 outlines the assumptions and operating environment for Mixclaves. These diverge from traditional mixnets and shape our goals for the platform. Section 3 describes the Mixclaves architecture, particularly how its design satisfies our goals from Section 2. Section 4 describes our prototype instantiation of the architecture, including caveats and practical details. Section 5 measures our prototype against our goals from Section 2. Section 6 contrasts the Mixclaves architecture with existing work before discussing future work and extensions in Section 7. Section 8 concludes.

2 MODEL AND GOALS

In the following sections, we provide a brief background on hardware enclaves. We then define and discuss the threat model and operating environment, with particular attention to points that distinguish our setting from traditional mixnets. We then enumerate the goals for Mixclaves in this context.

2.1 Enclaves

2.1.1 Enclave Overview. Secure enclaves are a hardware isolation boundary within a CPU that offers confidential computing (so that even privileged users cannot see inside) and memory encryption. They also support techniques to offer attestation, which is a process running inside the enclave that enables external systems to verify the identity and code running in the enclave [3].

Both AMD and Intel support enclaves in their modern server processors. Intel's SGX implementation runs individual binaries in enclaves. Its design encourages programs to run partially in the trusted execution area and partially in the normal computing environment. Then, information would pass between the two as

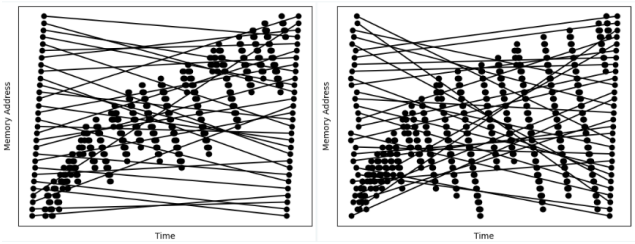


Figure 1: Message buffering access patterns in Loopix (left) vs Mixclaves (right). Loopix has a a correlation coefficient of 0.738 from input to output address accesses, whereas Mixclaves is 0.03.

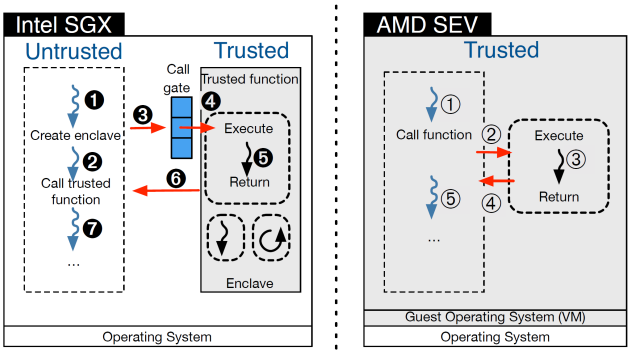


Figure 2: Intel SGX enclaves versus AMD SEV-SNP [25].

shown in Figure 2. In contrast, AMD's SEV approach runs an entire VM inside the enclave. Mixclaves are designed to operate almost entirely within the enclave; our architecture could apply to either setting.

We assume two properties of hardware enclaves. First, we assume the code executed inside the enclave is *attested* and both operators and clients can verify its integrity. Consequently, any code inside the enclave that should execute, will run; we assume a fail-stop model and exclude Byzantine mixnet nodes within a trust domain (Section 2.3). Second, we assume code removing a layer of encryption from the packet and examining its metadata inside the enclave leaks no information to an adversary, conceding that in the various enclave implementations this comes with caveats [33, 36, 46]; attested code executes confidentially, excluding side-channels discussed in Section 2.1.2.

2.1.2 Oblivious Data Structures. While enclaves offer stronger guarantees compared to traditional computing, there remain two attack vectors from an adversary with control over the network and underlying hardware: (1) monitoring network traffic in and out of the enclave and (2) monitoring memory access patterns. For the former, existing mixnets by design already prevent network observer attacks. However, they do not address memory access pattern leaks, so we investigate how to address this.

Because mixnet nodes store messages for a random delay before resending to the next hop, they need to buffer the messages in memory. This leads to a potential vulnerability with coordinating

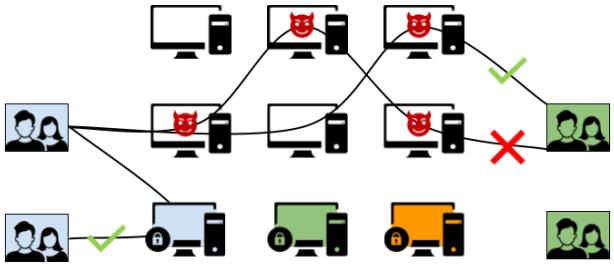


Figure 3: To preserve privacy, clients must route packets through at least one honest node or privacy may be compromised. Within a trust domain, it is sufficient to route packets through a single Mixclave node.

packets entering and leaving the enclave with memory access patterns. The buffering is taken care of in Twisted, an event-driven network engine that powers Mixclaves and the reference mixnet Loopix [38]. We examined the messages stored between processing and sending. Twisted appends new messages generated by a task to the end of an array. When the task completes, the array is sorted by its target execution time and the next task is dequeued from the front. Loopix and Mixclaves messages are delayed by a random value (see Section 4.1. This process rearranges some of the messages in the buffer, but there remains a strong, positive correlation between the input and output addresses (R=0.738), as shown in Figure 1. This is especially problematic if two users exchange significant traffic; then it becomes possible to correlate that the two are communicating, revealing metadata about the communication.

We altered Twisted's packet append operation to instead insert at a random location in the buffer before sorting. This modification obscures the original memory address of a particular message so that once it is accessed for sending, the packets leaving the enclave cannot be correlated to the original storage address. The correlation coefficient of store vs read memory addresses became R=0.03, as illustrated in Figure 1.

2.2 Mixclaves Threat Model

We assume our adversary can passively observe the entire network and memory traces on mixnet nodes. We further assume that an attacker can inject traffic into the network, also dropping, replaying, and delaying traffic arbitrarily. All information shared with clients is also known by the adversary. Mixclaves relies on the Sphinx packet format [17, 18] to provide bitwise unlinkability for messages passing through the mixnet. All Sphinx messages are padded to the same length and allow for detection of tagging attacks and replay attacks.

We assume a computationally restricted adversary, so the cryptography is sound and decrypting packets (particularly those chosen by the adversary) inside the enclave cannot leak information about the decrypted packet or any other traffic. As we discuss in section 2.1.2, delays chosen by an attacker must be obscured by Mixclaves when the message is queued.

We assume that any application over the mixnet generates sufficient cover traffic to conceal its workload between the mixnet and its endpoints. As in Groove [9] and Loopix [38], an application can deploy *provider* nodes that generate continuous, synthetic cover traffic if endpoints are only intermittently online. Similarly, any property of or over the data such as forward secrecy is maintained by the application. The mixnet also generates cover traffic between nodes, following an exponential distribution chosen by the operator [38].

Recall from Section 1 that traditional mixnets distribute trust in the mix network, relying on a sufficient fraction of honest nodes to preserve privacy. In contrast, Mixclaves adopt a *federated* trust model that partitions the network into independently operated *trust domains*. Each trust domain is mutually suspicious. Mix nodes and endpoints within a trust domain can verify the integrity of attested code on mix nodes, but not necessarily of mix nodes in other domains. For example, in Figure 3, Blue endpoints can be confident that the Blue mix node is honest, but can make no assumptions about Blue or Orange nodes beyond what is assumed in traditional mixnets. We discuss the consequences of a federated trust model in Section 2.3.

Finally, we assume that endpoints can learn the public keys not only for other endpoints, but also to at least one node in the active mixnet. Bootstrapping metadata-private communication is not necessarily out of band [31]. Section 3 describes how endpoints learn about the keys for previously unknown nodes in the mixnet.

2.3 Deployment Model

2.3.1 Distributed Trust. Relay networks like Tor [19] are deployed by volunteers, placing few or no restrictions on who may join the network. This *distributed* model for trust resists analysis by requiring an attacker to add malicious nodes until its targets route sufficient traffic through its network. The probability of clients choosing paths through not only corrupt nodes, but corrupt nodes controlled by a particular attacker is unknowable, but assumed to be sufficiently low that every path through the mixnet encounters at least one "honest" node, as in Figure 3.

In a distributed model, nodes may enter and leave without any controls or accountability. This affords operators of mix nodes full flexibility, but since the client chooses the path and mix nodes gain no information about the packets routed through it, operating mix nodes with an SLO entails an unbounded obligation to the network. Capacity planning for an application with no insight into the health of the network, no ability to predict how failures will impact traffic, and no identifying metadata is more art than science. As mentioned in Section 1, operating a mix net also carries liabilities [6, 24] that may deter providers from donating infrastructure to a mix network, even if the operator derives a fractional benefit by improving it.

Egalitarian architecture deployed on volunteer infrastructure succeeded spectacularly in the early evolution of the public internet. Today, the resources and expert knowledge necessary to operate a secure service— particularly one that draws the attention of state-sponsored surveillance— exceed the grasp of most enthusiasts.

2.3.2 Federated Trust. In contrast, Mixclaves build on a *federated* trust model that partitions mixnet nodes into *trust domains*. Nodes within a trust domain share a common operator interested in maintaining the health of the mixnet, but particularly for its subset of users. A trust domain is not a datacenter or region, but a set of

servers; one could even collocate mix nodes from different trust domains in the same datacenter. The full mixnet is a *federation* of trust domains. Trust within a domain is grounded in guarantees provided by the hardware enclaves (Section 2.2).

Where trust domains are managed by established providers, peering relationships could meter, manage, and price traffic through the federated mixnet. Endpoints could verify their right to access the mixnet in that domain without associating their identity [4, 21, 34], while traffic between domains could be shaped to do credible capacity planning. Aligning incentives for honest operation of trust domains is outside our scope, but we note that an operator has more flexibility for its subset of endpoints than in the distributed trust model. Section 2.3.3 highlights several contexts aligned with Mixclaves' trust model.

It must be acknowledged that trust is less diffuse in Mixclaves than in traditional mixnets; a compromised node in a trust domain may taint all the nodes in that domain. Endpoints benefit from the efficiency of shorter paths, but if the trust domain is compromised then clients must rely on attestation of audited code and the opacity of metrics available to the operator to preserve their privacy. This is also true of traditional mixnets, but Mixclaves not only exchange metrics among mix nodes in a trust domain, they also encourage users to accept even a single-node mix as sufficient.

To address the risk of correlated compromise, Mixclave networks adopt three remediations. First, Mixclaves support *anonymous administration* by processing control traffic obscured as mixnet traffic. Signed control packets can generate a response from within the enclave; the mix node collects statistics useful to an administrator (e.g., node queue lengths, CPU utilization) and construct a reply to a return address that can be routed back through the mixnet. Second, metrics reported should not leak sensitive information beyond what can be inferred by our threat model, so a compromised administrator cannot aid traffic analysis. Third, code run by the mixnode is attested by the enclave, so an administrator pushing new code to mix nodes can be detected.

We argue that a path through a single Mixclave node is sufficient to operate a differentially private mixnet. Applying oblivious extensions from Section 2.1.2, enclaves allow a mix node to *scale up* the anonymizing function that a *scale out* mix network provides. This leans heavily on the attestation and confidentially guarantees implemented in hardware enclaves.

2.3.3 Deployment Context. As we believe volunteer nodes joining a mixnet at random is not plausible for practical deployment, we instead envision that the edge is the more practical solution to build a mixnet with the federated deployment model of Mixclaves. The rise in the availability of this compute type is another reason why Mixclaves is conceivable as a design today.

Applications continue to move computation off end user devices to servers. These servers sometimes are positioned in the cloud, but more and more are pushed to the edge to reduce latency. Specific applications that benefit from this include multiplayer gaming, voice assistants, and autonomous vehicle systems, among others. The Cloudlet model, introduced by M. Satyanarayanan in 2008 [49], was composed of a vision for decentralized VM based Cloudlets to enable nearby computation offload for mobile devices. Modern edge computing has materialized in a different light, more in the flavor of colocation than in the original Cloudlet vision, but the importance of moving compute to many nodes has shown virtue. Mixclaves works well in the VM edge computing model as it lacks the need for any centralized cloud.

In a similar light, the rapid expansion of cloud and content providers (CCPs) has exposed the architectural deficiencies of the public Internet architecture for a service based economy. CCPs operate massive private networks to increase the performance available to their services. These manifest in two parts, a large backbone network (essentially a private WAN) that carries traffic between datacenters, and many user facing points of presence (PoPs) near to the clients. The PoPs intercept (and process) all traffic for a CCP, a process termed interposition.

The Extensible Internet (EI), proposed in [8], introduces interposition to the public internet with the deployment of service nodes (SNs). These allow execution of arbitrary services in the network, assumed to be backed with hardware enclave computing. Mixclaves already operates on a federated trust and inter-domain model, so a platform like EI makes the deployment of Mixclaves easier while maintaining the same guarantees of the current implementation.

2.4 Improving Performance via Differential Privacy

With Differential privacy, we can prove that we require only 1 mixclave node to provide strong privacy garuntees to the client. Differential privacy describes the promise from a data holder to a user that "You will not be affected, adversely or otherwise, by allowing your data to be used in any study or analysis, no matter what other studies, data sets, or information sources, are available" [22]. When applied to a mixnet, this means that the network guarantees, for a pair of users Alice and Bob, that the probabilities that the two are communicating or not communicating are close, as defined by the parameters ϵ, δ [30] (e^{ϵ} is a multiplicative factor, and δ is additive). So the users may statistically deny any communication has taken place.

Differential privacy is composable, so a user may reason about multiple cover stories. If the probability that Alice spoke with Bob is within (ϵ, δ) of her speaking to no one, which in turn is within (ϵ, δ) of her communicating with Charlie, there is a total probability difference of (2ϵ, 2δ). Based on the selection of parameters, a user may claim many different scenarios which are statistically probable.

Based on previous work in mixnets, a message must route through at least one honest node in the network to be differentially private [9, 30]. Consider a message sent between two clients, Alice and Bob. The probability that this condition fails to hold is the probability of meeting 0 honest nodes on a path of length L in the mixnet, i.e. BinomialCDF(0, L, p). And therefore the probability that the condition *does* hold is 1 - BinomialCDF(0, L, p). In previous work, the likelihood p of meeting an honest node has always been less than 1. But with Mixclaves, a client will always select an honest node, so p = 1. Therefore, Mixclaves always achieves differential privacy, even with a network as small as 1 node.

Mixclaves dramatically decrease the number of nodes needed to route between two clients while maintaining the same privacy. This also simplifies the threat model compared to those used in Groove or Loopix, as discussed previously.

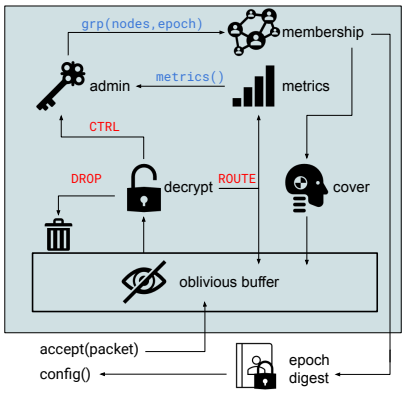


Figure 4: Mixclave mix node

2.5 Goals

Without compromising the privacy guarantees demonstrated in previous work, Mixclaves proposes to *improve the efficiency and operability* of mix networks.

Our principal metric is **cost** for delivering a target throughput for mix networks' opaque payloads at a latency comparable to a state-of-the-art, low-latency reference implementation. For a fixed number of nodes and target throughput, operating a Mixcalves mix network should be less expensive with comparable or lower latency. Since introspection of mix network traffic is impossible by design, we focus on aggregate throughput of saturating, low-latency packet flows.

To lower costs further, we extend the functionality of mix networks by proposing **anonymous administration**. Confidential compute and code attestation by hardware enclaves present an opportunity unavailable to earlier implementations. We apply these capabilities to add mix network **elasticity**. Elasticity can lower costs by adapting to changes in workload, rather than provisioning for peak throughput. We demonstrate that an operator can **monitor** nodes in a trust domain anonymously and use monitoring data to **reconfigure** the trust domain.

3 ARCHITECTURE

In this section we describe the Mixclave architecture in detail. Section 4 describes how our prototype implements this architecture to evaluate its practicality.

3.1 System Overview

A Mixclave node is composed of service modules, as shown in Figure 4. Nodes expose two public APIs, `accept(packet)` and `config()`. Sphinx packets routed to `accept` are inserted into an oblivious buffer inside the hardware enclave (Section 2.1.2). Once decrypted inside the enclave, DROP messages are discarded. Unsurprisingly, ROUTE messages are reinserted into the oblivious buffer to be forwarded after the user-configured delay. Cover traffic in Mixclaves are generated following an exponential distribution across all known mix nodes, as in Loopix [38].

What distinguishes a Mixclave node from a traditional mix node are the control packets (CTRL). These packets are signed by an administrator key recorded in the enclave. Control packets can coordinate updates to the trust domain membership, gather statistics, and change internal configuration state, like parameters for cover traffic. We discuss these in greater detail in the following sections. Once committed, the epoch digest is published through the `config` API.

3.2 Deployment and Operation

Mixclaves support *anonymous administration* by processing control packets signed by an operator within the enclave. Control packets can collect metrics, change settings, or add and remove nodes from the trust domain (reconfiguration, see Section 3.3). Control packets appear as normal mixnet traffic outside the enclave, but are distinguished in three ways.

First, control packets are **signed** by an operator whose public key is associated with permissions to the mixnet node. Our prototype does not partition operator capabilities, but one could separate monitoring, scaling, and reconfiguration across keys, or even include a "poison pill" that destroys the mixnode. Second, control packets may contain a **return address**. The mix node has the public key for the operator to verify the control packet, but it needs to know where to route the response. Not all control traffic generates a response. For example, one could implement a "warrant canary" packet that affirms the operator has *not* been forced to disclose information by means that are illegal for it to acknowledge [48]. Third, control packets may **read metrics** from nodes. Normal packets can push updates, but cannot read metric data. Control packets querying metrics are written and routed through the broader mix net back to the return address.

Control packets may be sent either by an operator or by other Mixclave nodes in the trust domain. Traffic loops could gather not only the round-trip time from a node, but also reliable observations from other nodes in the trust domain. Most importantly, all the operator traffic is concealed by the cover and real traffic to the enclave. Building a control plane in traditional mixnets would require one to adopt Byzantine protocols and the overheads of running at least $3f + 1$ control nodes, as in systems like Tapestry [50]. By adopting hardware enclaves and adopting a federated trust model, we can operate a system with similar or stronger guarantees at a fraction of the cost (see Table 1).

3.3 Reconfiguration

To meet our goals for elasticity, an operator (or an agent in an enclave) must be able to add and remove mix nodes. Since mixnets offer only best-effort delivery and persist no state, applications written for mixnets tolerate data loss; we are principally concerned with preserving anonymity during and after a reconfiguration. When nodes are added to a mixnet, cover traffic must include the new nodes before user traffic can be routed to it. In our architecture, an application is responsible for ensuring its own cover traffic was added to the mix before informing its clients, if necessary.

To see why this may be necessary, recall the provider nodes in the Loopix messaging service generate cover traffic in the mixnet. Before informing users, a Loopix service could first require consensus among its providers on the new nodes to ensure cover traffic includes them uniformly; mix nodes also generate noise, but they

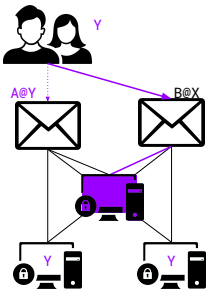


Figure 5: a) Messaging client C and provider A are at an epoch Y, provider B is at epoch X, $X \sqsubset Y$. C changes providers from A to B, but since B sends no cover traffic to the new node, traffic to N must be from C.

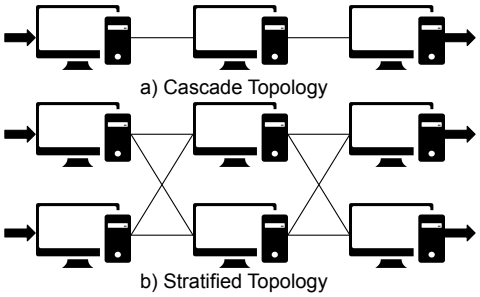


Figure 6: a) Chaum mixnet b) Loopix mixnet

cannot cover asymmetric traffic from application services. Consider the scenario in Figure 5 with no cover traffic. Assume the epoch changes from X to Y and Y adds a new node N. If a client changes providers from A at epoch Y to B at the earlier epoch X, then all traffic addressed to N from B will be client traffic[1]. Whether the cover traffic generated by the client and mixnet is sufficient is a choice for the application and trust domain operator and outside the purview of the mixnet.

Reconfiguration can be subtle [2], particularly when an operator supports concurrent reconfigurations from multiple operators or avoids pauses during reconfiguration (*online* reconfiguration). For example, virtual synchrony [11] maintains both group membership within a view and offers a reliable broadcast service, but typically pauses to install a new view. With a trust domain composed of widely distributed, small clusters of machines [49] may make different tradeoffs than one running in well-connected datacenters [8].

While we ruled out Byzantine behavior in the threat model from Section 2.2, attacks on hardware enclaves continue to evolve and could include a full architectural break. Even if an operator excluded the node, if a node can be prevented from shutting down, even when the node is isolated from the trust domain a client with an outdated view of the network could validate a compromised, Byzantine mix node. One solution could include a signed, timestamped lease recording the last contact with the operator. The client could use the lease to detect parts of the network abandoned by the operator or unreachable during reconfiguration[2], though rendering the domain unusable when an operator misses a check-in or asking a client to set a threshold for staleness both harm usability.

We conclude that selecting a particular reconfiguration algorithm entails material tradeoffs for a mixnet operator. In a federated model, the operator of a trust domain can select an algorithm for reconfiguration independently without affecting or informing the rest of the federation.

We do constrain reconfiguration by requiring that each mix node provide an API exposing the list of nodes, public keys, and optional node metadata with which it is configured. This *epoch digest* must be timestamped, versioned, and signed by the operator or infrastructure key that committed the reconfiguration. The epoch digest should also be signed by the mix node, including a nonce chosen by the client. Versions do not need to be totally ordered, but they must form a join semilattice such that concurrent reconfigurations have a deterministic merge function [2]. As a consequence, reconfigurations can never be retracted once committed; if an operator wants mix nodes to rejoin the mixnet after a crash (rather than as a new node with a new identity), the mix node must persist the digest before installing a new configuration. Epoch digests are public and can be used both by clients and also by other trust domains.

[1]There are many fixes the application could apply, such as B dropping routes to unknown mix nodes or forcing refresh at B, but both of these policies are affected by the implementation of mixnet reconfiguration in the trust domain.
[2]This could also serve as a "canary" for a compromised network.

4 IMPLEMENTATION

In this section we describe the prototype implementation of the architecture in Section 3 to inform the evaluation in Section 5.

4.1 System Overview: Loopix

Our prototype for Mixclaves extends Loopix [38]. Loopix is a message-based mixnet that adds a random delay to every layer of encryption to confound traffic analysis. The random delay is drawn from an exponential distribution. Loopix uses a *stratified topology* for its traffic, as shown in Figure 6. The mixnet is separated into layers such that each node is connected only to the mix nodes in adjacent layers; traffic flows in one direction.

Internally, Loopix uses the Twisted network engine [16] not only to schedule delivery of UDP packets, but also to schedule periodic system functions. As discussed earlier in Section 2.2, Loopix includes a *provider* node responsible for generating cover traffic and hosting mailboxes for intermittently connected users of its messaging service. Clients poll the provider at regular intervals for messages, retrieving authentic messages for the user, drop packets generated by other clients, and synthetic messages generated by the provider. When clients exchange messages, the path includes their respective providers on either side of the mixnet.

Mixclaves are intended as a general-purpose mixnet, so we focus our evaluation on its peak throughput, overheads on latency, and operating cost. To measure these across applications, we modify the Loopix implementation by merging its client and provider into a load generator that measures round-trip latency of self-addressed paths through a mixnet. Measuring Mixclave packet latency rather than Loopix message latency eliminates the client's polling loop its mailbox at the provider. We also eliminate application-specific noise generated by the provider so we can measure the aggregate

throughput of the network using only benchmark packets and intra-mixnet noise.

Our prototype does not enforce a topology among mix nodes among servers, so clients may choose any path. A malicious client could encode up to 20 hops before the Sphinx format cannot encode it given the target message size, but this is a mild optimization of a denial-of-service attack when the Sphinx library can generate 360M three-hop messages per second on a GCP `n2d-standard-2` machine. Since clients will choose short routes uniformly distributed among nodes in a trust domain, a fully connected topology allows Mixclaves to expand elastically and uniformly receive load as clients learn of new resources.

We modify the message buffer for the Twisted network engine as described in Section 2.1.2 so messages in the send buffer cannot be correlated. We did not implement a signature scheme for control packets, as the overhead of validation is negligible in our evaluation.

To demonstrate both anonymous administration and scaling, we implement an *updater* by creating a variant of the Loopix client. The updater samples metrics using the covert control plane described in Section 3. The updater is also responsible for informing the mixnet of a new epoch when nodes are added to the trust domain. Benchmark clients are informed by the mixnet via a polling loop. Anonymously gathering statistics, resizing the mixnet, and informing clients that refresh their topology to include new nodes is sufficient as a proof of concept of the scalable Mixclave architecture.

An *epoch* is a monotonically increasing counter used to identify the set of mix nodes in the trust domain. Nodes are added to the cluster in a particular epoch. The set of public keys for generating synthetic traffic and authenticating messages is stored locally in a database, versioned by epoch. In our prototype, the node starts with a database populated for its epoch. If the update fails to converge, the updater will refresh and retry adding its nodes at the next-highest epoch. Since the updater for our cluster is outside the mixnet, if it crashes during reconfiguration then it may leak resources. If the updater receives acknowledgements from all nodes, then it updater can commit the result by sending a message to the mixnet nodes. Once committed, any mixnet can publish the new set of nodes in that epoch. Any mix nodes that do not learn the epoch from the updater can learn that it is active when client traffic in that epoch is routed to it.

4.2 Cloud Deployment

Our prototype uses Docker [20] and Terraform [27] to automate deployment and the initial setup. Terraform not only provisions VMs for each entity, but it also configures a private overlay network (VPC) among the nodes in the mixnet. We also use Terraform to allocate new nodes and include them in the mixnet VPC. Docker adds some overhead that adversely impacts overall performance [13], but relative comparisons of mix nodes operating in and out of enclaves should be comparable. As shown in Figure 7, while these tools greatly simplified our packaging and deployment, they also added significant provisioning overhead and consequently, hysteresis to scaling decisions. We will explore strategies for reducing allocation overheads in future work.

4.3 Reconfiguration

Our prototype uses control packets to update mixnode membership. Rather than implementing reliable broadcast or distributed consensus, we demonstrate the control path using a simple gossip protocol. We implement the updater client described in Section 3.1 by polling the mixnet node(s) with a control packet querying metrics, including a return address for the updater. The node generates a reply packet with its current throughput using the updater's public key. We use the throughput measured in Section 5 to populate a table for scaling thresholds. If the throughput exceeds a threshold, the updater will allocate a new node to the mix net. Once configured, the updater generates a new database for the epoch and copies it to the mix nodes.

The updater then repeatedly sends control packets to mix nodes, instructing them to install the database assigned to that epoch. If the mix node is at an epoch below the threshold, it will attempt to load the database for that epoch. If successful, it generates a response packet to the updater. Clients query the epoch digest, which in our prototype is implemented using the same, straightforward notification protocol for the current epoch.

While straightforward, this reconfiguration procedure demonstrates that Mixclaves can pass information in and out of the enclave, use those data to make scaling decisions, and conceal its reconfiguration traffic with cover and benchmark traffic. Populating updates to the database with control packets, implementing signing and certificates, and fault-tolerant protocols we leave to future work.

5 EVALUATION

In this section, we measure the performance of Mixclaves and compare against our reference implementation. We investigated the performance of Mixclaves running in Google Cloud Platform (GCP). Each mixnode and load generator ran in its own VM. Except where explicitly noted, we use N2D instances in the `n2d-standard-2` profile with 2 vCPUs, 8 GB of RAM, and 10 Gbps of bandwidth. These machines are configured with AMD SEV-SNP enclaves.

Provisioning time not only from the provider but for our packaging is significant, even after tuning. Since these overheads are not critical for our evaluation, and general techniques for reducing these overheads are well-known [39], we pre-allocate a pool of VMs where elasticity applies. Our measurements varied significantly, but we show the rough breakdown of allocation cost in Figure 7.

Recall from Section 4.1 our benchmark client that creates self-addressed loops through the mixnet. Benchmark packets include an experiment identifier, timestamp, and sequence number. We draw the delay per layer of encryption from an exponential distribution around 1ms in all experiments unless noted. The client records the round-trip time (RTT) of these loops as the latency of the mix net, which should match applications' experience. Since latency is recorded by the same process, we avoid any issues with clock drift across machines. If the latency exceeds a configurable threshold then the packet is recorded as dropped. Late-arriving packets also report latency using the embedded timestamp.

5.1 Microbenchmarks

5.1.1 Enclave overhead. To demonstrate that the enclave would not become a bottleneck, we ran a network benchmark to saturation in

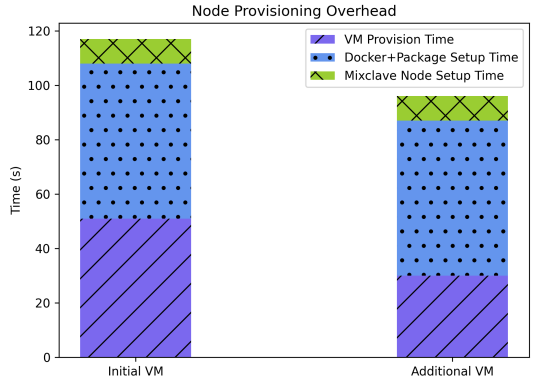


Figure 7: Breakdown of salient platform and framework provisioning costs.

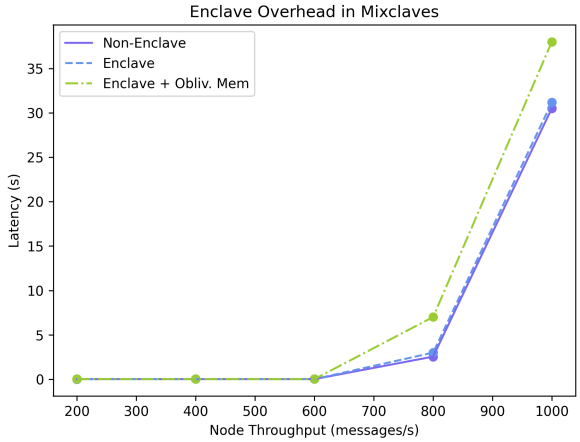


Figure 8: Throughput vs latency for a single node. Running in enclaves reduces peak throughput by around 200 messages per second.

and outside of enclaves using `iperf3`. We omit the graph for space, but the latency at lower throughput was indistinguishable and peak throughput saturated above 9.73Gbps outside the enclave and 9.55Gbps within it. This tranquilized any anxiety around the enclave implementation creating a network bottleneck for the Mixclave prototype. Docker did add a significant overhead for network traffic running in an enclave, reducing measured throughput to 4.98Gpbs.

5.1.2 Single Node. Figure 8 shows the throughput/latency for a single mix node running outside the enclave and within the enclave, with and without oblivious mitigation enabled (Section 2.1.2). The oblivious buffer adds overhead that lowers peak throughput per node. The enclave adds only a slight overhead for the low-latency workload, as expected.

5.2 Scalability

To evaluate the scalability of Mixclaves, we measure throughput of the mix network to saturation in multiple, fixed configurations, illustrated in Figure 9. The six-node mixnet (labeled "Loopix") records the round-trip latency through a three-hop network of non-enclave

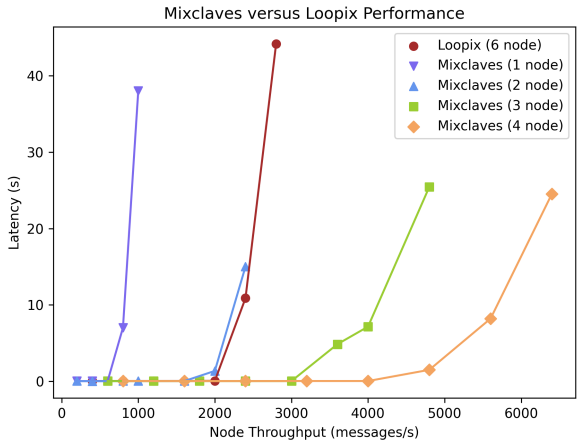


Figure 9: A network of six Loopix nodes saturates below 2400 messages per second (3 hops), but Mixclave cluster of three nodes saturates above 3000 messages per second at 54% of the cost.

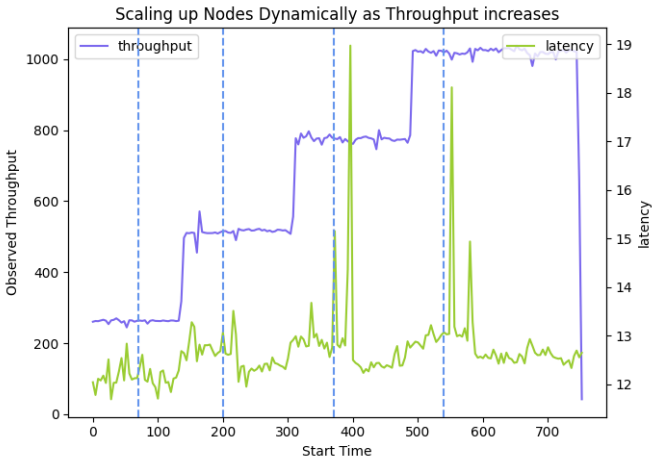


Figure 10: Adding nodes as throughput increases. A node is added to the mixnet at every dashed line, preserving latency.

machines. We increase the load on one through four Mixclaves nodes until saturation, recording the round-trip latency and with drop logic disabled. The oblivious buffer is disabled for the "Loopix" experiments, which run with the Mixclaves topology rather than the stratified topology of the original system.

Despite slightly lower performance per node, a Mixclave network supports *higher throughput at lower cost* than a multi-hop network. Decreasing packet path length more than overcomes the enclave overhead.

Mixnet	# Nodes	Peak Throughput (msg/s)	Monthly Cost ($)
Loopix	6	< 2400 / s	$296.04
Mixclaves	1	< 800 / s	$53.34
Mixclaves	2	< 2000 / s	$106.68
Mixclaves	3	< 3600 / s	$160.02
Mixclaves	4	< 4800 / s	$213.36

Table 1: Cost and throughput comparison of Loopix to two different Mixclave topologies.

5.3 Elasticity

To demonstrate support for reconfiguration (Sections 3.3 and 4.3), we gradually increase client traffic and show that Mixclaves throughput increases beyond saturation points measured in Section 5.2. As shown in Figure 10, during reconfiguration throughput sometimes fluctuates for reasons as yet unexplained, but it recovers quickly.

5.4 Cost

As of this writing, running a cloud VM in an enclave adds an 8% cost premium per node. Table 1 records the costs of operating a mix network on a fixed set of nodes, per month in GCP [40]. As shown in Figure 9, at or above the peak throughput the mix network saturates and drops traffic to avoid collapse. The figure also suggests savings available to an elastic network that can adapt to load, rather than provision for peak traffic.

Enclaves set a higher minimum price for compute resources. Public clouds offer less expensive VMs, particularly if an operator is willing to accept preemption of those resources in spot instances and preemptable VMs [40]. This is attractive as mix nodes persist no state, but an "honest" mix node cannot be fully stateless without accepting caveats for mix network availability. Frameworks rely on honest nodes to detect replay attacks, so if a node is intermittently available then an active attacker could bypass those protections. If the node changes its identity on restart, this is akin to failure and/or reconfiguration of the mix network (Section 3.3) with all its attendant complexity and impact on mix network throughput. The costs recorded in Table 1 reflect monthly costs for comparable service.

5.5 Reconfiguration

At our scale, reconfiguration of the mixnet is near-instantaneous after provisioning is completed. Dissemination of the updated epoch digest to clients will vary depending on the application, as discussed in Section 3.3. Our handful of benchmark clients usually noted and applied the new node configuration in less than a second, though recall from Section 4.3 that the updated epoch digest is already copied locally; the update packet only records which epoch the client should load for generating benchmark traffic.

Our experiments did reveal a challenge to concealing administrative traffic to *overloaded nodes*. Services often expose a high-priority channel reserved for administrative traffic— even running on a dedicated port— to ensure admin commands are processed promptly and in a coherent order. These strategies are unavailable in our setting, as admin traffic is designed to look identical to real traffic. Our updater resends the update message to all nodes until it receives an acknowledgment, but in heavily overloaded Mixnet networks with dropping enabled, some reconfiguration attempts still failed.

6 RELATED WORK

David Chaum [15] proposed *mixing networks* (mixnets) in 1981. This paper established the following criteria for anonymizing network traffic in a mixnet. First, an adversary cannot correlate packet payloads to infer a path between the source and destination on either side of the mixnet. In a Chaum mixnet, layered encryption removed at each node in the mixnet provides *bitwise unlinkability*; subsequent work [18] ensures that other attributes like packet size leak no signal to an adversary. Second, traffic analysis cannot reveal correlations in time between packet flows. In a Chaum mixnet, messages are batched at each node until a threshold, then released to a successor. Deduplication (resisting an attacker repeating a message into a mix) and batching create *anonymity sets* of packets; given a packet entering the mixnode, packets exiting the mixnode with a non-zero probability of being derived from that packet are in its anonymity set. While in Chaum mixnets that set is an explicitly signed batch, in later mixnets the probability of a packet being in that set may be a non-symmetric distribution[3]

Circuit mixnets like Real-Time Mixes [29], Vuvuzela [45], and Groove [9] route traffic along a consistent path, rather than routing every message independently. Circuits can also be used to build continuous, low-latency channels through a mixnet [29] for telecommunications. Similar to Tor [19], these metadata-private messaging systems connect endpoints using bidirectional circuits through the mixnet. Pairs of endpoints rotate circuits periodically based on a shared secret. In Groove, messages are not delivered directly between endpoints but rather to an intermediate *provider* node. Provider nodes solve the problem of intermittently-connected clients. Any application using a mixnet for anonymity must ensure that signal is buried in noise; little is accomplished if an attacker can observe multiple rounds where Alice sends exactly k messages into the mixnet and Bob subsequently fetches k messages. Providers not only generate synthetic traffic into the mixnet that simulates the application workload when clients are disconnected, they also ensure traffic between clients confounds traffic analysis.

Metadata-private messaging services like Groove group messages in rounds, ensuring that every circuit in the network has at least one message in each round. The 30 second message latency in Groove and other systems in its pedigree is not inherent to circuit mixnets, but an application decision. To ensure at least one honest node even in highly corrupt networks (20% adversarial), Groove routes can exceed 11 hops. Trust in enclaves simplifies our architecture and significantly reduces costs.

The Tor [19] network consists of a network of relay nodes and a directory service. At regular intervals on the order of minutes, users select a sequence of nodes from the directory based on configurable weights for desired bandwidth, accepted ports, and other criteria. The user creates a circuit through which their traffic is routed, with consistent entry and exit nodes. Tor has been vulnerable to analysis based on non-uniform packet sizes and timing attacks [7, 14]. Mixnets are designed to resist exactly this traffic analysis.

Loopix [38] is a low-latency mixnet that provides time independence using a per-layer delay chosen by the client. The delay is

[3]For example, if packet A arrives at t_A and B arrives at t_B, $t_A < t_B$, packets emitted between t_A and t_B are in A's anonymity set but not in B's. As $t_A \ll t_B$, the probability of B being in A's anonymity set diminishes.

chosen from an exponential distribution such that any packet entering the node since it was last empty is a potential member of its anonymity set. In practice this is vanishingly unlikely and the client chooses delays that satisfy low-latency service-level objectives (SLO) from the mixnet. Binary unlinkability is ensured by the Sphinx [18] packet format, also used by Mixclaves.

All of these systems [9, 14, 15, 19, 29, 38, 45] rely on at least one honest node in the mixnet.

One work in particular, "Towards User Privacy for Subscription Based Services" [10] builds an implementation named "Mixnet" that runs in trusted execution environments (Intel SGX), but this name is something of a misnomer. The system is a proxy service that scrambles user identities before connecting to a third party subscription service. While an interesting use case of enclaves, this work is unrelated to mixnets.

7 FUTURE WORK

Mixnet Reconfiguration. Our heuristics demonstrate that Mixclave networks can scale out based on confidential, intra-domain metrics, but these cover only a small corner of the workload and cost space. While a mixnet can safely add new nodes, we have not proven that shrinking the cluster is safe; an honest mixnet should generate plausible, diminishing cover traffic to ensure that not all packets arriving at the old address are real traffic. Given an algorithm to remove nodes, the workload could also scale vertically to different VM instances. We did not explore “scale up” regions of the cost space since Twisted makes limited use of multi-core processors.

Oblivious scaling. We did not explore scaling strategies that resist analysis. One would expect diurnal expansions and contractions of the mixnet, but expanding the size of the cluster out of phase could leak information about traffic if the scaling decision were not based solely on externally-visible data.

Health checks. Using intra-domain loops measuring round-trip time, nodes in a trust domain could gather information about the health of the broader mix network. This could not only inform clients of failed or slow nodes in the network, but also direct clients away from malicious or failed nodes while maintaining cover traffic [32]. Packet delivery through the mixnet is only best-effort, but the operator of a trust domain can improve reliability for its users with these metrics. Our analysis was limited to scaling based on traffic in a trust domain, but peering relationships between trust domains could even define SLAs for loop traffic that cannot be separated from real and cover traffic.

Multipath mixnets. All mixnet formats restrict packets to a single path through the network, as offering mixnodes a choice only helps corrupted nodes direct traffic to an adversary’s advantage. Instead, if a symmetric key were encrypted with the public keys of multiple nodes, a trusted Mixclave node could filter out nodes it suspects have failed. The list of nodes (even in other federations) is not confidential, but requiring the client to discover and resynchronize with the network state may be more brittle than granting the forwarding nodes the option. Within a trust domain, this technique could also load balance among mix nodes based on queue lengths [35].

Node labels. Vulnerabilities in hardware enclaves such as Plundervolt [36], Foreshadow [44], and ÆPIC Leak [12] undermine a core assumption of the Mixclave architecture. Since the Mixnode will faithfully and accurately report its configuration from the enclave, users may elect to route their traffic through a diverse set of architectures to avoid relying on a single vendor. We assume an attacker already has this information in our threat model, so node labels should grant an adversary no new advantages.

Utility noise. The cover traffic generated by modern mixnets [9, 38] is randomly generated. While Mixclaves only pass control traffic among trusted nodes amid synthetic and user traffic, the ability to examine metadata safely within the enclave also admits the possibility of self-addressed loops of *data* traffic, similar to Broadcast Disks [1]. Data loops could pass through non-enclave nodes and nodes in other trust domains, with random delays in minutes or hours. Re-purposing packets with high delays as the payload for cover traffic is also admissible; replicate a packet in the mixnet such that fragments arrive approximately when it should be forwarded. Given a time-to-live (TTL) and a forgiving retrieval SLO, one could provide an archive storage service to subsidize the cost and legitimize the purpose of an anonymizing mixnet.

Improved oblivious data structures. Metrics retained for mixnet monitoring could leak information about traffic patterns, if flushed to memory. Metadata derived from message flows within the enclave must resist analysis, though these techniques are beyond the scope of this paper. While our oblivious algorithm worked well for a low-latency workload with shallow queues, an adversary could fill the queue with high-delay packets. Since message dispatch does not require sorting but only oblivious sending of messages that reached their deadline, one could integrate more scalable oblivious data structures for the message queue [41].

8 CONCLUSION

This paper proposes *Mixclaves*, an enclave-backed mixnet for metadata-private communication. Mixclaves makes it easier to operate mixnets, requiring fewer nodes and enabling administrative scalability to be pushed into the network. It employs oblivious message buffering to mask memory access patterns at a node, enabling a mixnet to operate in a single enclave node while still maintaining differential privacy. This enables Mixclaves to transition to a federated trust model from a distributed trust model in previous mixnets. Mixclaves also enables elasticity by allowing the trusted enclave nodes to measure performance characteristics of the network and send control packets to signal a network expansion.

We implemented Mixclaves as an extension of Loopix, which serves as our reference benchmark. We compared the performance of both in VMs running in Google Cloud Platform, demonstrating Mixclaves does see an overhead of the enclave, but that the cost is lower and and overall bandwidth is higher in Mixclaves due to messages routing through less nodes with our architecture. We tested the control plane functionality of Mixclaves, and found that the network proactively scaled at runtime to accommodate higher throughput.

ACKNOWLEDGMENTS

We thank our colleagues whose experience and insight informed the design, implementation, and motivation for Mixclaves. Alphabetically: Emmanuel Amaro, Tiemo Bang, Natacha Crooks, Vivian Fang, John Kubiatowicz, Zhuohan Li, Mae Milano, Micah Murray, Aurojit Panda, and Scott Shenker.

REFERENCES

[1] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik. 1995. Broadcast disks: Data management for asymmetric communication environments. In *Mobile Computing*. Springer, 331–361.

[2] Marcos K Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. 2011. Dynamic atomic storage without consensus. *Journal of the ACM (JACM)* 58, 2 (2011), 1–32.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. ACM.

[4] Elli Androulaki, Mariana Raykova, Shreyas Srivatsan, Angelos Stavrou, and Steven M Bellovin. 2008. PAR: Payment for anonymous routing. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 219–236.

[5] Sebastian Angel and Srinath Setty. 2016. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 551–569.

[6] Jacob Appelbaum, Aaron Gibson, J. Goetz, V. Kabisch, L. Kampf, and Leif Ryge. 2014. NSA targets the privacy-conscious. https://daserste.ndr.de/panorama/aktuell/nsa230_page-1.html.

[7] Michael Backes, Aniket Kate, Sebastian Meiser, and Esfandiar Mohammadi. 2014. (Nothing else) MATor (s) Monitoring the Anonymity of Tor's Path Selection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 513–524.

[8] Hari Balakrishnan, Sujata Banerjee, Israel Cidon, David Culler, Deborah Estrin, Ethan Katz-Bassett, Arvind Krishnamurthy, Murphy McCauley, Nick McKeown, Aurojit Panda, Sylvia Ratnasamy, Jennifer Rexford, Michael Schapira, Scott Shenker, Ion Stoica, David Tennenhouse, Amin Vahdat, and Ellen Zegura. 2021. Revitalizing the Public Internet by Making It Extensible. *SIGCOMM Comput. Commun. Rev.* 51, 2 (may 2021), 18–24. https://doi.org/10.1145/3464994.3464998

[9] Ludovic Barman, Moshe Kol, David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2022. Groove: Flexible Metadata-Private Messaging. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 735–750.

[10] Allan Benelli. 2022. *Towards User Privacy for Subscription Based Services.* Master's thesis. ETH Zurich.

[11] Ken Birman and Thomas Joseph. 1987. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*. 123–138.

[12] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*. 3917–3934.

[13] Emiliano Casalicchio and Vanessa Perciballi. 2017. Measuring Docker Performance: What a Mess!!!. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (L'Aquila, Italy) *(ICPE '17 Companion)*. Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/3053600.3053605

[14] David Chaum, Farid Javani, Aniket Kate, Anna Krasnova, Joeri de Ruiter, Alan T Sherman, and D Das. 2016. cMix: Anonymization by high-performance scalable mixing. In *USENIX Security*.

[15] David L Chaum. 1981. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM* 24, 2 (1981), 84–90.

[16] Twisted Community. 2022. Twisted. https://www.twisted.org/.

[17] George Danezis. 2022. The Sphinxmix python package. https://github.com/UCL-InfoSec/sphinx.

[18] George Danezis and Ian Goldberg. 2009. Sphinx: A compact and provably secure mix format. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 269–282.

[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. *Tor: The second-generation onion router.* Technical Report. Naval Research Lab Washington DC.

[20] Inc Docker. 2022. Docker. https://docker.com.

[21] Cynthia Dwork, Moni Naor, and Amit Sahai. 2004. Concurrent zero-knowledge. *Journal of the ACM (JACM)* 51, 6 (2004), 851–898.

[22] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.* 9, 3–4 (aug 2014), 211–407. https://doi.org/10.1561/0400000042

[23] Lee Ferran. 2014. Ex-NSA Chief: 'We Kill People Based on Metadata'. https://abcnews.go.com/blogs/headlines/2014/05/ex-nsa-chief-we-kill-people-based-on-metadata. (2014).

[24] Eva Galperin, Kurt Opsahl, and Dia Kayyali. 2014. Dear NSA, Privacy is a Fundamental Right, Not Reasonable Suspicion. https://www.eff.org/deeplinks/2014/07/dear-nsa-privacy-fundamental-right-not-reasonable-suspicion.

[25] Christian Gottel, Rafael Pires, Isabelly Rocha, Sebastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. 2018. Security, Performance and Energy Trade-Offs of Hardware-Assisted Memory Protection Mechanisms. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. IEEE.

[26] David Green and Katitza Rodriguez. 2014. NSA Mass Surveillance Programs: Unnecessary and Disproportionate. *Electronic Frontier Foundation* (2014).

[27] HashiCorp. 2022. Terraform. https://www.terraform.io/.

[28] Apple Inc. 2022. https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf. (Accessed 2022-12-13).

[29] Anja Jerichow, Jan Muller, Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. 1998. Real-time mixes: A bandwidth-efficient anonymity protocol. *IEEE Journal on Selected Areas in Communications* 16, 4 (1998), 495–509.

[30] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 711–725.

[31] David Lazar and Nickolai Zeldovich. 2016. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 571–586.

[32] Hemi Leibowitz, Ania M Piotrowska, George Danezis, and Amir Herzberg. 2019. No right to remain silent: isolating malicious mixes. In *28th USENIX security symposium (USENIX security 19)*. 1841–1858.

[33] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium (USENIX Security 21)*. 717–732.

[34] Li Lu, Jinsong Han, Yunhao Liu, Lei Hu, Jin-Peng Huai, Lionel Ni, and Jian Ma. 2008. Pseudo trust: Zero-knowledge authentication in anonymous P2Ps. *IEEE Transactions on Parallel and Distributed Systems* 19, 10 (2008), 1325–1337.

[35] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[36] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1466–1482.

[37] Trevor Perrin and Moxie Marlinspike. 2016. The double ratchet algorithm. *GitHub wiki* (2016).

[38] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The loopix anonymity system. In *26th USENIX Security Symposium (USENIX Security 17)*. 1199–1216.

[39] Google Cloud Platform. 2022. Guides: Create custom images. https://cloud.google.com/compute/docs/images/create-custom.

[40] Google Cloud Platform. 2022. VM Instance Pricing. https://cloud.google.com/compute/vm-instance-pricing. Accessed 2022-12-09.

[41] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2565–2579.

[42] Johns Hopkins Foreign Affairs Symposium. 2014. The Price of Privacy: Re-Evaluating the NSA. https://youtu.be/kV2HDM86XgI?t=1073.

[43] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 423–440.

[44] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient {Out-of-Order} Execution. In *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[45] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 137–152.

[46] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power {Side-Channel} Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*. 679–697.

[47] WhatsApp. 2016. WhatsApp Encryption Overview. https://www.whatsapp.com/security/WhatsApp-SecurityWhitepaper.pdf.

[48] Wikipedia. 2022. Warrant Canary. https://en.wikipedia.org/wiki/Warrant_canary.

[49] Adam Wolbach, Jan Harkes, Srinivas Chellappa, and M. Satyanarayanan. 2008. Transient Customization of Mobile Computing Infrastructure. In *Proceedings of the First Workshop on Virtualization in Mobile Computing* (Breckenridge, Colorado) *(MobiVirt '08)*. Association for Computing Machinery, New York, NY, USA, 37–41. https://doi.org/10.1145/1622103.1622108

[50] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. 2004. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications* 22, 1 (2004), 41–53.