

Making Data Storage Efficient in the Era of Cloud Computing

Yang Tang

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2020

ABSTRACT

Making Data Storage Efficient in the Era of Cloud Computing

Yang Tang

We enter the era of cloud computing in the last decade, as many paradigm shifts are happening on how people write and deploy applications. Despite the advancement of cloud computing, data storage abstractions have not evolved much, causing inefficiencies in performance, cost, and security.

This dissertation proposes a novel approach to make data storage efficient in the era of cloud computing by building new storage abstractions and systems that bridge the gap between cloud computing and data storage and simplify development. We build four systems to address four data inefficiencies in cloud computing.

The first system, GRANDET, solves the data storage inefficiency caused by the paradigm shift from upfront provisioning to a variety of pay-as-you-go cloud services. GRANDET is an extensible storage system that significantly reduces storage costs for web applications deployed in the cloud. Under the hood, it supports multiple heterogeneous stores and unifies them by placing each data object at the store deemed most economical. Our results show that GRANDET reduces their costs by an average of 42.4%, and it is fast, scalable, and easy to use.

The second system, UNIC, solves the data inefficiency caused by the paradigm shift from single-tenancy to multi-tenancy. UNIC securely deduplicates general computations. It exports a cache service that allows cloud applications running on behalf of mutually distrusting users to memoize and reuse computation results, thereby improving perfor-

mance. UNIC achieves both integrity and secrecy through a novel use of code attestation, and it provides a simple yet expressive API that enables applications to deduplicate their own rich computations. Our results show that UNIC is easy to use, speeds up applications by an average of 7.58 \times , and with little storage overhead.

The third system, LAMBDATA, solves the data inefficiency caused by the paradigm shift to serverless computing, where developers only write core business logic, and cloud service providers maintain all the infrastructure. LAMBDATA is a novel serverless computing system that enables developers to declare a cloud function’s data intents, including both data read and data written. Once data intents are made explicit, LAMBDATA performs a variety of optimizations to improve speed, including caching data locally and scheduling functions based on code and data locality. Our results show that LAMBDATA achieves an average speedup of 1.51 \times on the turnaround time of practical workloads and reduces monetary cost by 16.5%.

The fourth system, CLEANOS, solves the data inefficiency caused by the paradigm shift from desktop computers to smartphones always connected to the cloud. CLEANOS is a new Android-based operating system that manages sensitive data rigorously and maintains a clean environment at all times. It identifies and tracks sensitive data, encrypts it with a key, and evicts that key to the cloud when the data is not in active use on the device. Our results show that CLEANOS limits sensitive-data exposure drastically while incurring acceptable overheads on mobile networks.

Contents

List of Figures	iv
List of Tables	ix
Acknowledgments	xi
Chapter 1 Introduction	1
1.1 Paradigm Shifts in the Era of Cloud Computing	1
1.2 Challenges and Opportunities	4
1.3 Hypothesis	5
1.4 Overview of Contributions	6
1.5 Dissertation Organization	9
Chapter 2 A Unified, Economical Object Store for Web Applications	11
2.1 Background: Cloud Storage Services	15
2.2 Extended Motivation and Example	18
2.3 Architecture	21
2.4 Deciding Data Object Placement	27
2.5 File System Interface	30

2.6	Implementation and System Extensibility	32
2.7	Evaluation	35
2.8	Discussion	47
2.9	Related Work	48
2.10	Summary	50
Chapter 3 Secure Deduplication of General Computations		52
3.1	Security Model and Design	57
3.2	UNIC API and Usage	61
3.3	Leveraging Storage Deduplication	66
3.4	Implementation	67
3.5	Evaluation	70
3.6	Discussion and Limitations	81
3.7	Related Work	83
3.8	Summary	84
Chapter 4 Optimizing Serverless Computing by Making Data Intentsexplicit		85
4.1	Background: Serverless Computing	88
4.2	A Motivating Example	91
4.3	LAMBDATA Overview	100
4.4	Data-Aware Scheduling	104
4.5	Optimization: Direct File Access	107
4.6	Evaluation	108

4.7	Discussion	118
4.8	Related Work	120
4.9	Summary	121
Chapter 5 Limiting Mobile Data Exposure with Cloud-based Idle Eviction		122
5.1	Case Study: Data Exposure on Android	126
5.2	Goals and Assumptions	131
5.3	The CleanOS Architecture	134
5.4	Prototype Implementation	143
5.5	Applications	149
5.6	Evaluation	152
5.7	Security Discussion and Limitations	163
5.8	Related Work	165
5.9	Summary	168
Chapter 6 Conclusion		169
Bibliography		171

List of Figures

Figure 2.1	<i>Monthly cost with (a) variable object size and (b) variable number of requests. Each line corresponds to a storage service. Assuming (a) has fixed 100 GET requests and (b) has fixed 1MB object size. Each request counts as one EBS I/O.</i>	18
Figure 2.2	<i>CDF of file size for Piwigo.</i>	19
Figure 2.3	<i>GRANDET deployment scenarios.</i>	22
Figure 2.4	<i>GRANDET components. Components with a plug-in symbol mean that developers can easily extend them.</i>	23
Figure 2.5	<i>An example of PUT request and response.</i>	24
Figure 2.6	<i>GRANDET's PHP SDK and usage examples.</i>	27
Figure 2.7	<i>Implementation of GRANDET's file system interface. Solid arrows are the data flow. Dashed arrow shows the logical relationship between the file structure and its content.</i>	31
Figure 2.8	<i>GRANDET's Actor class.</i>	33
Figure 2.9	<i>Comparing the cost of different backends and GRANDET. All costs are normalized to the optimal cost.</i>	38
Figure 2.10	<i>Number of objects at each backend over time.</i>	40

Figure 2.11	<i>Performance of GRANDET backend.</i> Evaluated with 4KB and 1MB requests. Error bar is standard deviation.	42
Figure 2.12	<i>GRANDET's end-to-end performance.</i> Time is normalized to the baseline which uses EBS directly.	43
Figure 2.13	<i>Scalability of GRANDET when using single S3 storage.</i> Evaluated with requests of 4KB in size. The error bar is the standard deviation.	45
Figure 2.14	<i>End-to-end scalability.</i> Evaluated on the FileSender application with real workload.	45
Figure 3.1	<i>UNIC protocol.</i> We use as the concatenation operator.	60
Figure 3.2	<i>A simple virus scanning application.</i>	62
Figure 3.3	<i>First step: memoize the computation result.</i>	63
Figure 3.4	<i>Final version: use filesystem metadata to further reduce I/O operations.</i> .	64
Figure 3.5	<i>UNIC architecture.</i> Additional hosts each have the same architecture as Host 1, and are omitted here due to limited space.	67
Figure 3.6	<i>Misalignment between line and chunk boundaries in grep.</i> Shaded region is the adjusted chunk for computation.	72
Figure 3.7	<i>Throughput of put() and get() operations.</i> The x-axis is the size of memoized result. The y-axis is the total time in performing 10,000 put() (solid line) and get() (dashed line) operations.	74

Figure 3.8	<i>Relative running time of applications.</i> The y -axis is the running time relative to the original application. For each cluster, the first bar is cache-miss execution without FS deduplication, the second bar is cache-hit execution without FS deduplication, the third bar is cache-miss execution with FS deduplication, and the fourth bar is cache-hit execution with FS deduplication. The dashed line at 100% shows the running time for the original application.	76
Figure 3.9	<i>Effectiveness of memoization with evolving data.</i> Solid line is the original grep without memoization. Dashed line has the result cache populated with v3.0. Dotted line has the result cache populated with the immediate previous version.	78
Figure 3.10	<i>Effectiveness of memoization across users.</i> For each cluster, the first bar is the original application, and the second bar is the application modified to use UNIC. Each bar shows the breakdown of running time on each group, the number on top showing the total time.	79
Figure 4.1	<i>Example of specifying data intent for a cloud function that generates the thumbnail of an image.</i> It reads input from <code>pic/1.jpg</code> and writes output to <code>thumb/1.jpg</code>	86
Figure 4.2	<i>Overview of serverless architecture.</i>	89
Figure 4.3	<i>A photo-sharing application example.</i> Solid arrows indicate triggers. Dashed arrows represent data flows.	91
Figure 4.4	<i>Example code with four cloud functions.</i>	92

Figure 4.5	<i>Data flow of handling user upload.</i>	93
Figure 4.6	<i>Data flow of making collage.</i>	93
Figure 4.7	<i>Inefficient scheduling on OPENWHISK.</i>	96
Figure 4.8	<i>Dependency between collage and thumbnail.</i>	99
Figure 4.9	<i>Overlapping functions in a pipeline.</i>	99
Figure 4.10	<i>A rare case of overlapping functions in a pipeline.</i>	99
Figure 4.11	<i>Optimized scheduling with LAMBDATA.</i>	100
Figure 4.12	<i>Example annotations for an invocation of collage.</i>	102
Figure 4.13	<i>LAMBDATA's architecture. Components with italic font are LAMBDATA-specific.</i>	103
Figure 4.14	<i>Microbenchmark: median time to get and put objects of various sizes to Amazon S3. Lower is better.</i>	109
Figure 4.15	<i>Time breakdown and speedup of the thumbnail function for various image sizes.</i>	114
Figure 4.16	<i>Timeline for two representative workloads.</i>	116
Figure 5.1	<i>The CleanOS Architecture. Key components are highlighted in grey. We add or modify in some way all of the boxed components (except for FS and kernel).</i>	135
Figure 5.2	<i>The CLEANOS SDO API and device-cloud protocol.</i>	138
Figure 5.3	<i>CleanOS Taint Tag Structure. We impose a structure on TaintDroid taints to support arbitrary numbers of taints.</i>	145
Figure 5.4	<i>Screenshot of Audit Service Log in App Engine.</i>	151

Figure 5.5	<i>Audit precision.</i> Each bar shows the average probability over time that tainted data was actually exposed, given that the audit log shows its SDO as exposed.	155
Figure 5.6	<i>Micro-operation Performance (milliseconds).</i> CleanOS Java object field access times compared with Android, TaintDroid. Times for non-sensitive and sensitive fields for various eviction states. Averages over 1,000 accesses.	156
Figure 5.7	<i>Application Performance.</i> Performance of various popular app activities under Android, TaintDroid, and CleanOS for various eviction states and configurations. Results are averages over 40 runs, in milliseconds.	157
Figure 5.8	<i>Energy Consumption.</i> Hourly energy consumption attributed by PowerTutor to the three apps when running a long-term synthetic workload for at least 3 hours over Wi-Fi. Numbers on top of each bar show energy overhead over default Android in percent.	161
Figure 5.9	<i>Network Traffic Patterns of Apps vs. CleanOS.</i> CleanOS traffic vs. app traffic for a one-hour trace. The Y axis is in log scale. In our cases, the phone has background traffic, which is included in both app and CleanOS lines.	163

List of Tables

Table 2.1	<i>Overview of AWS storage services (May 2016).</i>	16
Table 2.2	<i>Approximate monthly price (US dollars) for select AWS storage services (May 2016).</i> [†] S3 infrequent access charges a minimum of 128KB storage for smaller objects.	17
Table 2.3	<i>List of studied web applications.</i>	19
Table 2.4	<i>Lines of code of GRANDET’s components.</i>	32
Table 3.1	<i>Lines of code changed for each application.</i> Parenthesis indicates whether the adaptation uses file-level or block-level memoization. The numbers for gcc are based on ccache.	71
Table 3.2	<i>Storage overhead.</i> Columns are: (a) the number of input files, (b) total size of input files, (c) number of entries in the result cache, (d) size of the Redis dump file, and (e) relative storage overhead.	81
Table 4.1	<i>Inputs and outputs of each function.</i>	94
Table 4.2	<i>Time spent on each phase of the functions, in milliseconds.</i> The last column shows the percentage of time spent on getting the data from the cloud storage.	97

Table 4.3	<i>List of all functions and the parameters used in the experiment.</i>	111
Table 4.4	<i>Breakdown of the phases in each function.</i> All times are in milliseconds. The last column shows the speedup with cached data.	113
Table 4.5	<i>Description of two representative workflows.</i>	115
Table 4.6	<i>Monetary cost.</i> Numbers are in $\times 10^{-6}$ dollars.	118
Table 5.1	<i>CLEANOS Modifications to Android, TaintDroid.</i>	125
Table 5.2	<i>Examples of captured sensitive data.</i>	127
Table 5.3	<i>Exposure of cleartext sensitive data across all 14 apps.</i> A ‘Y’ indicates that we obtained cleartext copies from RAM/DB. A ‘-’ does not mean that the data is not on the device, but just that we did not find it in cleartext; the data could exist in some encrypted form.	128
Table 5.4	<i>Examples of when hoarded sensitive data is being actually used by the apps.</i>	129
Table 5.5	<i>Sensitive data exposure period.</i> Numbers are the fraction of time in which sensitive data was exposed.	153
Table 5.6	<i>Sensitive data lifetime.</i> Numbers are the maximum sensitive data reten- tion period.	154

Acknowledgments

The work presented in this dissertation would not have been possible without the help and support of many people.

In particular, I express my greatest gratitude to my advisor, Professor Junfeng Yang, for his advice and guidance over the past nine years. I would not have been able to complete this work without his support, encouragement, and inspiration.

I thank Professor Roxana Geambasu for her useful suggestions on the projects I worked on and the presentations I gave at conferences.

I thank Professor Vishal Misra for his helpful comments on my thesis proposal.

I thank Petros Maniatis and Xi Wang for their valuable feedback on parts of the work that contributed to this dissertation.

Last but not least, I thank my colleague students Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Heming Cui, Gang Hu, Nikhil Sarda, Lingmei Weng, Jingyue Wu, and Xinhao Yuan, who helped me solve lots of technical issues and proofread many of my manuscripts.

This dissertation is dedicated to my wife, Yuan Du, for her constant love and unconditional support, to my daughter, Karen Tang, for all the joy she brings, and to my parents, Yansheng Tian and Lei Yang, for their encouragement throughout my life.

Chapter 1

Introduction

1.1 Paradigm Shifts in the Era of Cloud Computing

We enter the era of cloud computing in the last decade, when people are changing ways to write and deploy web applications. Four major paradigm shifts are happening in terms of cost, resource-sharing, management, and mobility. They bring many benefits to the developers.

1.1.1 Cost

Traditionally, developers have to provision and pay for all resources they intend to use upfront, including computing instances, data storage, and network infrastructures. It is hard to estimate the right amount of resources, causing over- or under-provisioning. Over-provisioning wastes money, and under-provisioning may lead to service outage or degradations. Worse, if the workload changes dynamically, the developer has to provision for the peak scenario, letting resources idle at other times.

In the era of cloud computing, developers can choose from a variety of cloud offerings, paying only for what they use. This pay-as-you-go model brings great flexibility to the developers and eases their burden of upfront costs. They can scale up or down

dynamically based on the current workload.

As a result, many companies are adopting cloud computing to cut costs. A 2016 survey of 500 business and IT executives reports that over 40% of companies say the benefit of cutting costs outweighs all other benefits of cloud computing [33]. Goldman Sachs reports that by moving to the cloud, it has increased productive capacity by 580% while reducing the total costs by 20% [24].

1.1.2 Sharing resources

Traditionally, developers keep all infrastructures on-premises. They own and control all the computing and storage resources, and do not share them with other parties.

In the era of cloud computing, resources are shifting from single-tenancy to multi-tenancy. Mutually distrusting cloud users are now running computations on shared instances and storing data on shared storage. It improves resource utilization and reduces redundancy. For example, a 2009 study on 162TB disks shows that sharing data can reduce storage consumption by 68% [92]. Microsoft reports that sharing computation results can eliminate 35,000 hours of redundant computations per day [69].

1.1.3 Management

Traditionally, developers not only program their business logic, but also have to manage their infrastructures, including both software (*e.g.*, operating systems and libraries) and hardware (*e.g.*, machines, disks, and network infrastructures).

In the era of cloud computing, the burden of maintaining infrastructures gradually

shifts to cloud providers. The paradigm first shifts to Infrastructure-as-a-Service, where cloud providers manage the hardware and developers manage the software. In recent years, serverless computing emerges as a new paradigm to build cloud applications. Developers write small functions that react to cloud infrastructure events, while the cloud provider maintains all resources and schedules the functions in containers. Thus, developers only focus on their core business logic, and leave server management and scaling to the cloud providers.

As a result, many companies, including Netflix, Coca-Cola, and the New York Times, are adopting serverless computing [35]. A 2018 survey of 600 IT decision-makers shows that 61% of respondents are already using or plan to use serverless computing by 2020 [57].

1.1.4 Mobility

Traditionally, desktops and servers are the primary computing platforms. They are immobile and connected via wired networks. Mobile phones have few functionalities besides making phone calls.

In the era of cloud computing, mobile technology is developing at the same time. Smartphones are replacing desktops as the primary personal computing platform. Network connectivity becomes ubiquitous so that smartphones are always connected to the cloud.

Some 2017 statistics show that mobile devices generate 67% of all web visits in the United States [50]. As a result, many developers in the era of cloud computing adopt a mobile-first strategy.

1.2 Challenges and Opportunities

Although cloud computing brings many benefits, one big challenge is that data storage abstractions have not changed much for many decades. The conventional way of handling data in the era of cloud computing is inefficient. The lack of evolution in data storage abstractions turns each benefit of cloud computing in §1.1 into great challenges.

First, instead of upfront provisioning, cloud providers offer a variety of cloud storage services in a pay-as-you-go manner. Unfortunately, these services have incompatible interfaces, such as filesystems and key-value stores. They also have different durability, availability, and latency guarantees. Worse, their complex pricing schemes make reasoning about storage costs difficult. Developers face great challenges in choosing from so many cloud storage options. Many developers store their data inefficiently on the cloud, causing poor performance and large bills. This challenge gives us an opportunity to build a unified and economical cloud data storage.

Second, the multi-tenancy nature of cloud computing means that a significant portion of the data is redundant. Data deduplication can hugely save storage and simplify management. However, not only is the data redundant, but the computations on top of the data are also redundant. Unfortunately, cloud users do not trust one other. It is challenging for cloud users to share computations with other cloud users that they do not trust. Traditional deduplication techniques fail to work because they only focus on trusted data. This challenge gives us an opportunity to deduplicate computations for mutually distrusting cloud users.

Third, in the new paradigm of serverless computing, developers only write core busi-

ness logic as cloud functions, while the cloud provider maintains all the infrastructure. However, cloud functions are ephemeral, with no persistent storage at all. It is challenging to share data efficiently across cloud functions in serverless computing. Most developers resort to cloud storage systems for persisting states. However, this practice is an inefficient compromise suffering from long latency and high cost, because cloud storage systems are not designed with serverless computing in mind, and their evolution lags behind this paradigm shift. Specifically, cloud functions are blind to which data they read or write, therefore missing potentially huge optimizations. This challenge gives us an opportunity to design a new data option for serverless computing.

Fourth, smartphones are replacing desktop computers as people's primary computing platform. They are prone to losses, yet people store enormous sensitive data there, such as credit card numbers and private communications. How to manage sensitive data on mobile devices remains challenging because their operating systems are ported from desktop computers and are not designed with physical insecurity in mind. If the device is stolen or lost, all sensitive data are at risk. This challenge gives us an opportunity to evict idle sensitive data to the cloud since smartphones are almost always connected.

1.3 Hypothesis

The paradigm shift to cloud computing brings many benefits, but the lack of evolution in data storage abstractions causes many inefficiencies:

1. It is difficult to choose from too many cloud storage options.
2. It is hard to deduplicate computations for mutually distrusting cloud users.

3. Cloud functions cannot share data directly.
4. Mobile devices mismanage sensitive data.

Therefore, we propose the following research hypothesis: we can make data storage efficient in the era of cloud computing by building new storage abstractions and systems that bridge the gap between modern cloud computing techniques and legacy data storage solutions, and in the meantime, simplify development.

1.4 Overview of Contributions

In order to address the challenges presented in §1.2 and solve the inefficiencies defined in §1.3, we have built four systems, forming the four main contributions of this dissertation:

1. **GRANDET**: a unified, economical object store for web applications.
2. **UNIC**: secure deduplication of general computations.
3. **LAMBDATA**: optimizing serverless computing by making data intents explicit.
4. **CLEANOS**: limiting mobile data exposure with cloud-based idle eviction.

We now briefly overview each system.

1.4.1 **GRANDET**

Our first contribution, **GRANDET**, solves the data storage inefficiency caused by the paradigm shift from upfront provisioning to a variety of pay-as-you-go cloud services.

GRANDET is a novel, extensible storage system that significantly reduces storage costs for web applications deployed in the cloud. It serves as a layer between web applications

and cloud storage. GRANDET provides both a file system interface and an S3-like key-value interface, supporting a broad spectrum of web applications. Under the hood, GRANDET supports multiple heterogeneous stores and unifies them by placing each data object at the store deemed most economical. Specifically, for each supported store, GRANDET maintains a profile capturing the store’s pricing model, availability, durability, and consistency guarantees, and performance such as latency. It updates the performance part of this profile by periodically running its *profiler*, and the other parts based on crawling or user-supplied configurations. Given a data object, GRANDET runs its *predictor* to predict the future workload on the object, and its *decider* to determine, on a fine-grained, per-object basis, the most economical store that meets the default or developer-specified quality of service (QoS) requirements—even the default is better than the typical web practice. GRANDET preserves the availability, durability, and consistency that the cloud stores provide. When the workloads or pricing models change, GRANDET migrates data objects automatically as needed to reduce costs. We explicitly designed GRANDET to be extensible so that developers can add new stores easily.

1.4.2 UNIC

Our second contribution, UNIC, solves the data inefficiency caused by the paradigm shift from single-tenancy to multi-tenancy.

UNIC is a novel system that securely deduplicates general computations. It serves as a memoization layer that allows applications running on behalf of mutually distrusting users to memoize and reuse computation results, thereby improving performance.

UNIC achieves both integrity and secrecy through a novel use of code attestation, a classic primitive to attest what code is running to a (remote) party [125, 124]. To insert or query the result cache that UNIC maintains, UNIC generates a secure, non-forgable key that attests to both the application code and the input data. This key strongly isolates applications from each other in the result cache.

UNIC provides a simple yet expressive API that enables applications to deduplicate their own rich computations. From a high level, this API supports an application to (1) insert $input \rightarrow result$ to the result cache UNIC maintains, and (2) query the cache with $input$ and get back the cached $result$ if any.

1.4.3 LAMBDATA

Our third contribution, LAMBDATA, solves the data inefficiency caused by the paradigm shift to serverless computing, where developers only write core business logic, and cloud service providers maintain all the infrastructure.

LAMBDATA is a novel serverless computing system that adds a cache layer between cloud functions and cloud storage, enabling developers to declare a cloud function’s data intents, including both data read and data written. Once data intents are made explicit, LAMBDATA performs a variety of optimizations to improve performance and reduce costs.

Operationally, LAMBDATA works as follows. It leverages GRANDET or existing cloud object storage (e.g., AWS S3) to store data. LAMBDATA adds a caching layer, where each computing node has its own object cache. LAMBDATA schedules cloud functions based on both code and data locality. It tends to schedule multiple function invocations working

on the same data on the same computing node, so they can reuse cached data.

1.4.4 CLEANOS

Our fourth contribution, CLEANOS, solves the data inefficiency caused by the paradigm shift from desktop computers to smartphones always connected to the cloud.

CLEANOS is a new Android-based mobile operating system, which adds to the mobile device a cloud-based data management layer that manages sensitive data rigorously and maintains a clean environment at all times in anticipation of device theft. The crucial insight in CLEANOS is to leverage the tight integration of today’s mobile applications with trusted cloud-based services in order to evict sensitive in-memory and on-disk data to those services whenever it is not needed on the device. CLEANOS identifies and tracks sensitive data in memory and on stable storage, encrypts it with a key, and evicts that key to trusted cloud storage when the data is not in active use on the device. CLEANOS thus ensures that the minimal amount of sensitive data is exposed on the vulnerable device at any time.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents GRANDET, a unified economical object store for web applications. Chapter 3 presents UNIC, a system that securely deduplicates general computations. Chapter 4 presents LAMBDATA, a server-less computing system optimized by making data intents explicit. Chapter 5 presents CLEANOS, a mobile operating system that limits data exposure with cloud-based idle evic-

tion. Chapter 6 concludes this dissertation.

Chapter 2

A Unified, Economical Object Store for Web Applications

Web applications are getting more ubiquitous every day because they offer many useful services to consumers and businesses. Examples include Instagram and Flickr for hosting, processing, and sharing images; YouTube and Vimeo for videos; Pandora and Spotify for music; and Dropbox and Google Drive for files.

Many of these web applications can become quite storage-intensive. At the initial deployment of these applications, a single server might be enough to host the data from their limited number of users. However, as they become more successful, hosting images, videos, files, and other data objects from millions of users, their storage needs increase dramatically. For instance, Facebook has over 500 million users with 260 billion images, totaling 20PB [17]. Dropbox has over 50 million users, storing 500 million files daily [41].

Cloud computing provides an attractive, economical choice for meeting the storage (and computational) needs of web applications. Besides the usual benefits of elastic scaling and no hardware (over-)provisioning, each cloud platform typically supports a range of storage options with different performance, durability, and price characteristics. For instance, Amazon Web Services (AWS) supports non-persistent virtual disks (*instance store*), persistent virtual disks (*elastic block store*, or EBS), and key-value object store (*simple storage service*, or S3). Each of these options typically has more sub-options, such as EBS on

SSD or magnetic disks, and S3 with reduced redundancy or infrequent access. This rich set of choices gives developers the flexibility to pick the best one that meets their applications' needs. Unsurprisingly, most web startups today choose to deploy their apps in the cloud, so that they can focus their scarce manpower and funding on features of their applications [8].

Unfortunately, despite all these storage options, it remains quite challenging for developers to best leverage them to minimize cost. For simplicity in programming, it is common practice for a developer to pick a store she thinks is the best and places all objects of the same data collection (*e.g.*, all images) within the store. However, at its core, minimizing cost requires developers to make fine-grained decisions on *which* store is the best for *which* object. The reason is that the pricing models of different stores are quite complex and subtle, depending on such factors as the size of the object, the number of various types of access requests, and the direction and amount of network transfers. Two objects in the same data collection may differ hugely regarding these factors, and therefore should be placed at different stores. Consider two AWS stores, EBS on SSD, which charges a high price for storage and nothing for requests, and S3, which charges a moderate price for both storage and requests. A large but cold (*i.e.*, few read and write requests) object should be stored in S3, whereas a small but hot object should be stored in EBS. It is both non-intuitive and impractical to require developers, especially those at startups with scarce manpower and funding, to make such fine-grained placement decisions on a per-object basis.

In addition, many of the factors affecting price are highly dynamic, frequently requiring data objects to be migrated from one store to another to minimize cost. For instance,

the hotness of an object varies over time; so the best store for the object now may be the worst fit in the future. Even the pricing models change over time due to technology improvements [25] and competitions [15]. It is impractical to require developers to predict these changes accurately or migrate data objects manually.

Lastly, different stores provide heterogeneous interfaces, and a web application written against one storage interface (*e.g.*, the file system interface) may not be able to use another more economical storage option easily or at all. Many popular web applications, such as MediaWiki (the most popular wiki app) and WordPress (the most popular blogging app), still store data objects such as images in file systems. To run these applications in the cloud without significant modifications, developers have to store the data objects, however large they are, in a file system on top of EBS, an option potentially much more expensive than storing the objects in S3. While newer web applications tend to adopt S3, they may still manipulate the data objects using existing utilities that require the file system interface. Examples include a photo gallery using ImageMagick to process images or generate thumbnails, a video sharing application using ffmpeg to convert video formats, and a file sharing application using bzip2 to compress files. Thus, developers have to move the objects explicitly between S3 and the file system. These movements, if frequent, are not only complex to program but also expensive to execute, because S3 charges for both requests and network transfers.

Because of these reasons, it is difficult for developers to place objects optimally for minimizing cost. The cost of misplacement can be quite high. At a micro level, each PUT request on S3 costs as much money as storing 5MB of data for a day; so it is extremely costly to store frequently accessed data objects on S3. The storage cost on EBS is up to 8×

as much as on S3; so putting an infrequently-accessed large object on EBS is expensive, too. At a macro level, our experiments show that misplacement costs up to 572% more.

We present GRANDET, an extensible storage system that significantly reduces storage costs for web applications deployed in the cloud. GRANDET provides both a file system interface and an S3-like key-value interface, supporting a broad spectrum of web applications. Under the hood, GRANDET supports multiple heterogeneous stores and unifies them by placing each data object at the store deemed most economical. Specifically, for each supported store, GRANDET maintains a profile capturing the store’s pricing model, availability, durability, and consistency guarantees, and performance such as latency. It updates the performance part of this profile by periodically running its *profiler*, and the other parts based on crawling or user-supplied configurations. Given a data object, GRANDET runs its *predictor* to predict the future workload on the object, and its *decider* to determine, on a fine-grained, per-object basis, the most economical store that meets the default or developer-specified quality of service (QoS) requirements—even the default is better than the typical web practice. GRANDET preserves the availability, durability, and consistency that the cloud stores provide. When the workloads or pricing models change, GRANDET migrates data objects automatically as needed to reduce cost. We explicitly designed GRANDET to be extensible so that developers can add new stores easily.

We implemented GRANDET in AWS and evaluated GRANDET on a diverse set of four popular open-source web applications, namely CumulusClips, Piwigo, Elgg, and File-Sender. Our results show that:

1. GRANDET significantly reduces the costs spent on storage for web applications. On

average, GRANDET reduces the storage costs by 42.4%.

2. GRANDET has little overhead. It can be deployed with little impact on application performance.
3. GRANDET scales well when the workload increases.
4. Web applications can use GRANDET to save cost with no modification at all, and several lines of changes would reduce the cost even further.

The remainder of this chapter is organized as follows. The next section introduces the background of cloud storage services. §2.2 extends our motivation with a study and an example. §2.3 describes GRANDET’s architecture. §2.4 shows the data placement strategy. §2.5 presents the file system interface. §2.6 describes the implementation. §2.7 shows evaluation results. §2.8 discusses some design implications, §2.9 presents related work, and §2.10 summarizes this chapter.

2.1 Background: Cloud Storage Services

The variety of cloud storage options can be mainly divided into two categories: file storage and blob storage. File storage generally provides a disk or file system interface. Applications can mount it and manipulate data using file system operations such as `open()`, `read()`, and `write()`. Examples of file storage are Amazon elastic block store (EBS), Microsoft Azure file storage, and Google compute engine persistent disks. On the other hand, blob storage generally provides a minimal key-value interface, such as `PUT`, `GET`, and `DELETE`. A blob is normally treated as a whole, and operations such as partially up-

Storage service	Type	Durability	Availability	Latency
Instance store	file	ephemeral	99.95%	lowest
EBS (SSD)	file	99.8-98.9%	99.999%	lowest
EBS (magnetic)	file	99.8-98.9%	99.999%	low
S3 (standard)	blob	$1 - 10^{-11}$	99.99%	medium
S3 (reduced)	blob	99.99%	99.99%	medium
S3 (infrequent)	blob	$1 - 10^{-11}$	99.9%	medium
Glacier	blob	$1 - 10^{-11}$	n/a	high

Table 2.1: Overview of AWS storage services (May 2016).

dating a blob are absent. Examples of blob storage are Amazon simple storage service (S3), Microsoft Azure blob storage, and Google cloud storage.

Even more options are available for each category of cloud storage. For instance, Amazon Web Services (AWS) supports four types of stores (see Table 2.1). *Instance store* provides free, non-persistent virtual disks to an AWS elastic compute cloud (EC2) instance. These virtual disks are non-persistent because they are stored in the physical disks of the host machine that happens to run the EC2 instance. *Elastic block store (EBS)* provides persistent virtual disks, based on either SSD or magnetic. *Simple storage service (S3)* is a key-value store for objects, with standard, reduced-redundancy, or infrequent-access options. *Glacier* is a backup store with an extremely low cost and long read latency (3–5 hours).

Not only do these storage options have different service levels, but they also have complex and diverse pricing models. A typical pricing model depends on the total storage size, the number of each type of request, and the direction and amount of network data transfer. Table 2.2 shows a snippet of the pricing scheme for AWS storage services. Although they have the same data transfer cost, the discrepancies in storage pricing are

Storage service	Storage (/GB)	Request (/million)		Transfer (/GB)	
		PUT	GET	In	Out
EBS (SSD)	0.1	0	0	0	0.09
EBS (magnetic)	0.05	0.05	0.05	0	0.09
S3 (standard)	0.03	5	0.4	0	0.09
S3 (reduced)	0.024	5	0.4	0	0.09
S3 (infrequent)	0.0125 [†]	10	1	0	0.09
Glacier	0.007	50	50	0	0.09

Table 2.2: *Approximate monthly price (US dollars) for select AWS storage services (May 2016).* [†]S3 infrequent access charges a minimum of 128KB storage for smaller objects.

up to an order of magnitude, and those in request pricing can be as large as three orders of magnitude. No option is cheaper across all dimensions. For example, EBS (SSD) does not charge for I/O requests, but its storage price is more than three times as high as S3. By contrast, S3, despite charging less for storage, has high per-request cost.

To further illustrate the pricing discrepancies, let us study how much money it costs to put one data object on each of these storage services. Figure 2.1 shows the cost with (a) variable object size and (b) variable number of requests. We exclude data transfer cost in the figures for better clarity because it is the same for all these services. In each figure, the optimal choice is the minimum of all lines (shaded), and the threshold points are marked. We can see that the optimal choice depends on both object size and the number of requests, let alone each choice also has different durability, availability, and latency.

Thus, the heterogeneity of service levels and pricing schemes lead to extremely tough decisions that web applications should make when using cloud storage services. Misplacing data at non-optimal storage locations may not only cause service degradation but also cost a lot of money, negating the benefits that the cloud brings.

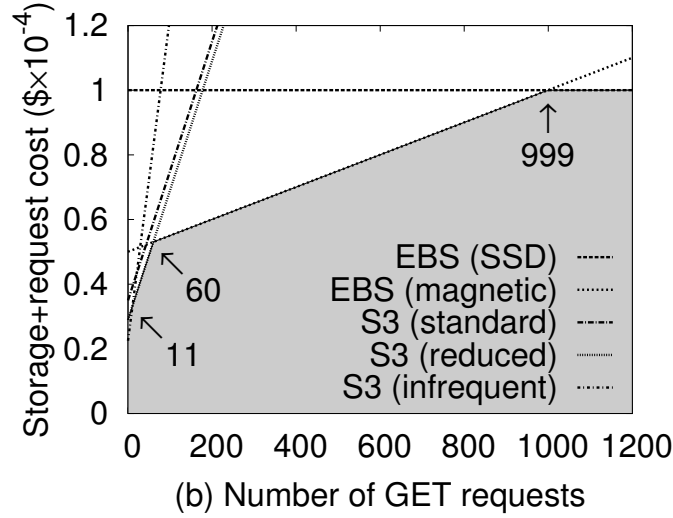
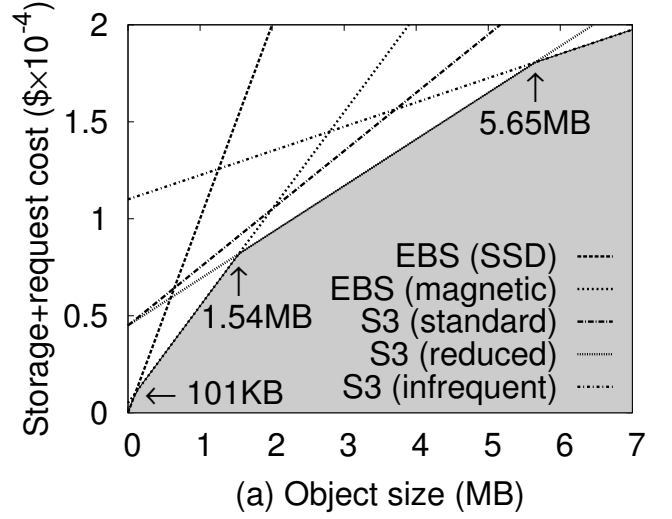


Figure 2.1: *Monthly cost with (a) variable object size and (b) variable number of requests.* Each line corresponds to a storage service. Assuming (a) has fixed 100 GET requests and (b) has fixed 1MB object size. Each request counts as one EBS I/O.

2.2 Extended Motivation and Example

We motivated the design of GRANDET by studying 19 popular open-source web applications of various kinds, including file sharing, photo and video sharing, shopping, blogging, news-reading, social networking, wiki, and content management systems (see Table 2.3 for the list). We observed two insights from our study.

Web application	Category	Web application	Category
FileSender	file sharing	selfoss	RSS reader
Piwigo	photo sharing	Tiny Tiny RSS	RSS reader
OpenPhoto	photo sharing	Elgg	social network
CumulusClips	video sharing	MediaWiki	wiki
OpenCart	shopping	LionWiki	wiki
PrestaShop	shopping	Wikka	wiki
Zen Cart	shopping	Drupal	CMS
Wordpress	blog	October	CMS
NibbleBlog	blog	Anchor	CMS
Chyrp	blog		

Table 2.3: *List of studied web applications.*

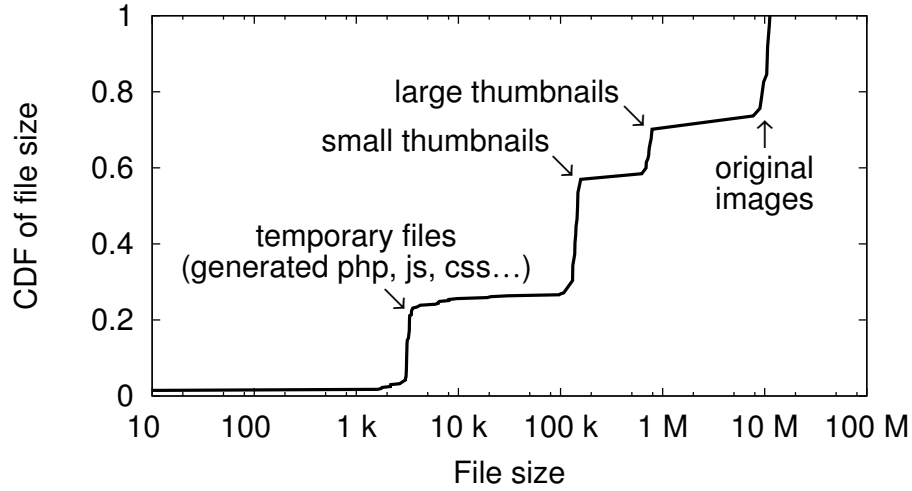


Figure 2.2: *CDF of file size for Piwigo.*

Our first insight is that data files have diverse yet clustered sizes and access patterns. For example, the original photo or video files are large, while the thumbnails are small. Additionally, some files are frequently read, such as a celebrity’s photo, while other files stay cold after they are stored, and the access pattern of files may change over time. For example, Figure 2.2 shows the distribution of file size for the Piwigo photo sharing application from our evaluation workload based on real-world statistics (see §2.7 for workload details). About 30% of all files are original images (7–12MB), 13% are large thumbnails

(600–800KB), 30% are small thumbnails (90–160KB), and the rest are temporary files. The reason that there are fewer large thumbnails is that Piwigo generates them lazily, and many photos are not accessed yet. This diversity creates a great opportunity for optimization, because file size and access pattern are two dominating factors on storage costs, as we have shown in §2.1.

Our second insight is that, despite their complexity, all the 19 applications manipulate data files only in simple ways. Each file corresponds to a logical data object, such as a photo or a video. These files are written sequentially and free of sub-file updates. Therefore, both file storage and blob storage are capable of storing these data objects.

Because of these two insights, we design GRANDET as a transparent gateway for a variety of heterogeneous storage services. Data objects are always stored at the optimal service based on the characteristics of the data and workload as well as the pricing and network condition. They are also automatically migrated among the storage services when the workload, pricing, or network condition changes. Next, we present a motivating example about how the CumulusClips video sharing application [39] stores and uses data, to illustrate how GRANDET can help it reduce storage cost.

When a user uploads a video file, CumulusClips stores it to the file system and calls an external program, `ffmpeg`, to convert the file to multiple formats, such as a high-definition version for broadband connections and a low-definition version for mobile devices. It also generates a static thumbnail of the video. All these derived files are stored in the file system, too. Later, viewers of the CumulusClips website see a list of thumbnails. When the viewer clicks on a thumbnail, based on her platform, one of the converted videos is played.

GRANDET helps CumulusClips by transparently handling the storage for all files. Despite internally storing data as key-value objects, GRANDET is mounted to CumulusClips’s uploads directory as a file system, and no modification to CumulusClips’s source code is required. Whenever CumulusClips wants to write a file to the directory, GRANDET puts the file to its optimal storage service based on its prediction of the file’s workload. For example, it would put a small thumbnail on EBS if it predicts that the file would frequently be read but put a large high-definition video on S3. GRANDET also migrates data over time to reflect latest conditions. For example, if an unknown video on S3 suddenly becomes a sensation (the “slashdot effect”), then GRANDET would move it to EBS for cheaper request cost.

2.3 Architecture

We now give an overview of GRANDET’s deployment scenarios and present the architecture of GRANDET.

2.3.1 Overview

GRANDET unifies multiple heterogeneous cloud storage into a single service. Its primary goal is to reduce storage costs for web applications. Thus, instead of running as standalone servers that would incur additional cost, GRANDET leverages piggyback deployment.

Figure 2.3 shows two typical deployment scenarios. For single-instance web applications, the GRANDET service simply co-locates on the same machine with the application (Figure 2.3(a)). Large-scale web applications (*e.g.*, MediaWiki [143]) typically shard their

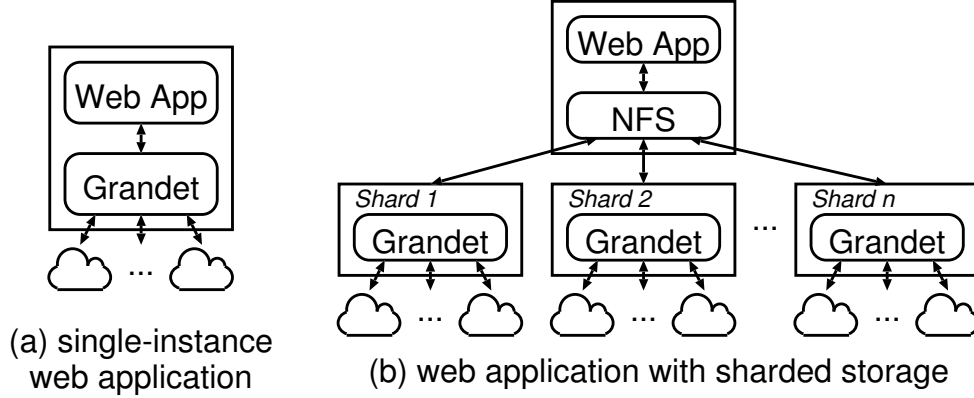


Figure 2.3: *GRANDET deployment scenarios.*

files into multiple storage servers, each storing a disjoint subset of the files, and mount them via a distributed file system (*e.g.*, NFS). In this case, each shard independently runs a GRANDET service on it (Figure 2.3(b)). Because the files stored on each shard are disjoint, GRANDET need not worry about consistency among shards.

GRANDET does not introduce new availability, durability, or consistency concerns due to two reasons. First, each object is stored on exactly one cloud storage; so the availability, durability, and consistency of GRANDET’s storage are as good as the underlying cloud storage. The application developer can specify the minimum availability, durability, and consistency requirement on a per-object basis (see §2.3.3). Second, since the GRANDET service itself resides on the same server as the web application or the storage shard, they share the same availability.

2.3.2 GRANDET components

Figure 2.4 shows the components of the GRANDET service. GRANDET’s frontend exports a key-value SDK for various programming languages and a general file system interface

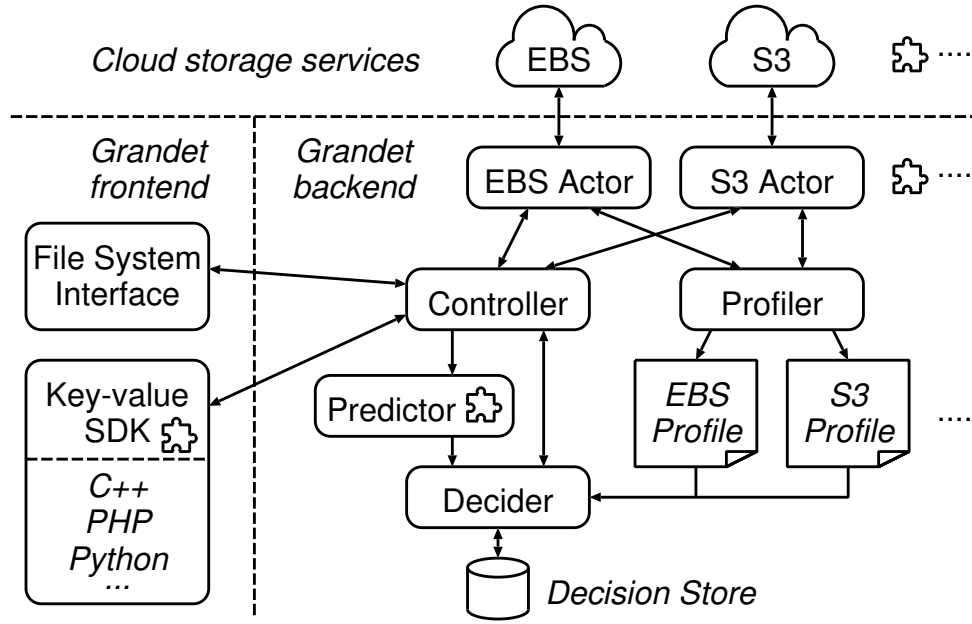


Figure 2.4: *GRANDET* components. Components with a plug-in symbol mean that developers can easily extend them.

to the web application or storage shards. The frontend and the backend communicate via Unix domain socket IPC.

The *GRANDET* backend stores data as key-value objects. It consists of five components. The Controller handles all requests from the frontend and coordinates all the other backend components. A set of Actors executes storage actions on a variety of storage backends. The Profiler periodically probes the current pricing model and network conditions for each storage backend, and stores them as *profiles*. The Predictor keeps track of the frequency of all PUT, GET, and DELETE requests, and predicts future request patterns. The Decider decides upon the best storage option based on the application’s requirements, the predicted request pattern, and the storage profiles. Decisions are kept on the *decision store* in Redis [116] and further persisted on EBS or S3.

We specifically designed *GRANDET* to be extensible, so that developers can easily add

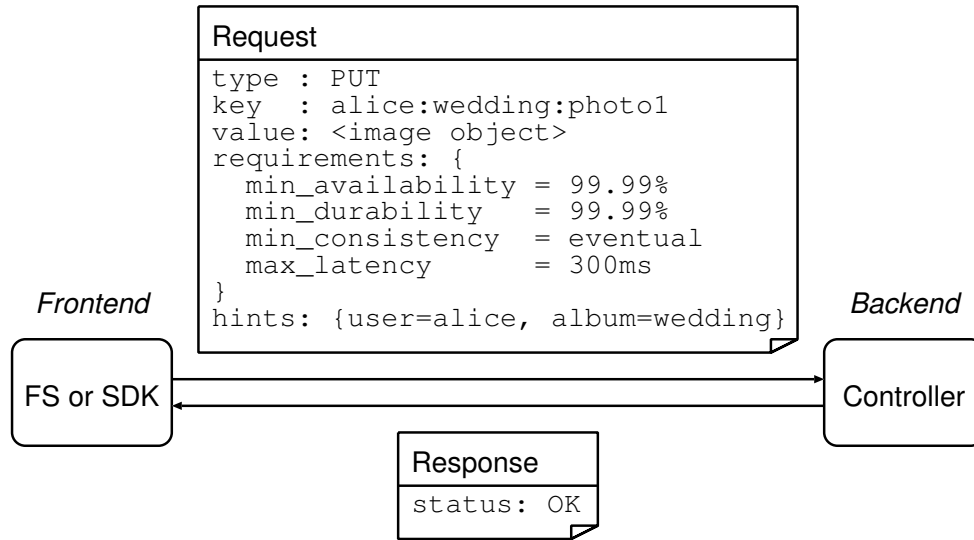


Figure 2.5: An example of PUT request and response.

new stores, support new languages, or change the prediction algorithm (see §2.6).

2.3.3 GRANDET workflow

All communications start with the application¹ sending a request to the Controller by using either the key-value SDK or the file system interface, where the latter internally represents files as key-value objects (see §2.5). Regardless of frontend, the request is one of the following:

PUT. The application requests to store a data object to GRANDET’s storage (Figure 2.5). The application should assign a unique *key* to the data object based on its own needs. For example, a photo sharing application may assign the image file that Alice uploads to her Wedding album the key `alice:wedding:photo1`. The *value* of the data object can be an arbitrary length of binary content.

¹If deployed with sharded storage, it is actually the shard that sends the request. However, there is no difference from GRANDET’s view.

Along with the PUT request the application can specify its *requirements* on the storage service for this particular data object. The requirements include the minimum availability, durability, and consistency required, as well as the maximum latency allowed. Only the services that meet these requirements are considered as candidates for storing this data object (see Table 2.1 for an overview of storage services). Requirements are optional. If the application does not specify requirements, then GRANDET assumes all non-ephemeral (*i.e.*, not the EC2 instance store) and moderate-latency (*i.e.*, not Glacier) services can be chosen. It is worth mentioning that even this default assumption provides better guarantees than a typical application’s setup, since both EBS and S3 are at least 20× more reliable than typical commodity disks [6].

The application can also give *hints* to GRANDET for a better placement decision. Hints are also optional, and we have implemented two default hints. §2.4 discusses the placement strategy and default hints in detail.

Upon receiving the request, the Controller first asks the Decider for the placement decision, which in turn looks at the current profile for each storage service and asks the Predictor for the predicted future request pattern. Based on this, the Decider finds the most cost-effective storage choice that satisfies all the application’s requirements, memorizes the choice at the decision store, and returns the choice to the Controller. Then the Controller tells the corresponding Actor to store the data object to the actual storage and notifies the Predictor to bookkeep this action. Finally, it tells the application that the PUT has completed.

GET. The application requests to retrieve a data object from GRANDET’s storage. The

Controller asks the Decider to recall the previous placement decision from the decision store and then asks the corresponding Actor to retrieve the data object from the actual storage. The Controller also asks the decider to check if the optimal placement decision would change because the current workload, pricing scheme, and network conditions may have changed. If not, it notifies the Predictor to bookkeep this action and returns the data object to the application. Otherwise, it also migrates the data object to the new storage service and deletes the old copy.

DELETE. The application requests to delete a data object from GRANDET's storage. The Controller asks the Decider to recall the previous placement decision from the decision store and then asks the corresponding Actor to delete the data object from the actual storage. It also notifies the Predictor to bookkeep this action.

2.3.4 Frontend interface

GRANDET has two types of frontend interface. The key-value SDK provides bindings for these requests for various programming languages such as C++, PHP, and Python. For instance, Figure 2.6 shows GRANDET's PHP interface and examples of putting and getting an image. The interface is similar to current cloud blob storage services such as S3. Therefore, web applications that are already aware of S3-like blob storage can just switch to GRANDET's SDK and seamlessly get all the cost-savings that GRANDET brings.

For applications that only work with file systems, GRANDET also provides a file system interface using FUSE, which applications can mount to their data directory directly. §2.5 describes it in detail.

```

// PHP SDK interface
function put($key, $value, $requirements=[], $hints=[])
function get($key)
function del($key)

// Example: PUT an image with requirements and hints.
require_once 'grandet.phar';
grandet\put('alice:wedding:photo1', $uploaded_image,
    ['min_availability_required' => 99.99,
     'min_durability_required'   => 99.99,
     'min_consistency_required' => 'eventual',
     'max_latency_required'     => 300],
    ['user' => 'alice', 'album' => 'wedding']);

// Example: PUT with no requirements and default hints.
grandet\put('alice:wedding:photo2', $another_image);

// Example: GET an image.
$image = grandet\get('alice:wedding:photo1');

```

Figure 2.6: *GRANDET's PHP SDK and usage examples.*

2.4 Deciding Data Object Placement

The cornerstone of GRANDET is the decision engine for placing each data object onto the optimal storage service. It makes a decision each time the application PUTs or GETs a data object. The decision engine closely follows the pricing model of all storage options. As mentioned in §2.1, a typical pricing model consists of three factors: storage (data size and lifetime), request (type and amount), and data transfer. The data size is known, and transfer prices are usually the same for all services within the same cloud region. Therefore, the key to making placement decision is predicting the future access pattern of the data object.

2.4.1 Prediction of access pattern

GRANDET provides a framework that allows developers to use any algorithm to predict access patterns (see §2.6.2). It also provides a default predictor, which we now describe.

For each request for a certain object, the predictor uses the request’s metadata to classify the object into the class of objects similar to this object. The metadata includes the object size, the object name, the requirements of the request, and other hints (see §2.4.3) provided by the developer.

For each class, the predictor keeps track of the number of GET and PUT requests issued on the objects in this class recently, and it also records the number of recently accessed objects and the average lifetime of the objects in this class. Each record is kept for r seconds (typically a day or a week).

Suppose that for the class the current object belongs to, there are g GET requests and p PUT requests recently, and there are n objects accessed in this class, then the predictor would predict that in the following t seconds, there would be $\frac{gt}{nr}$ GET requests and $\frac{pt}{nr}$ PUT requests for this object. It would also predict the object’s lifetime to be the average object lifetime in its class. Note that even for objects in the same class, their final storage decisions may differ, because the numbers (g, p, n) are dynamic and other factors such as object size may be different.

This default predictor is simple yet effective in our evaluation (see §2.7). We believe that recent machine learning techniques may empower even better algorithms, which can be easily plugged into GRANDET (detailed in §2.6.2).

2.4.2 Decision making

GRANDET's decider works with the predictor to decide where to place the object. It uses the object size and the predicted access pattern to make the decision. For each backend, GRANDET's decider uses its pricing model to calculate the storage cost of the object in its predicted lifetime and chooses the backend with the lowest cost.

The optimal placement decision for an object may change over time because of changed workload, pricing scheme, or network condition. A migration happens on a PUT or GET request when the extra cost for migration is less than the cost savings at the new storage service.

The extra cost for a GET-triggered migration is the total cost of an additional PUT request, a DELETE request, and data transfer cost, while a PUT-triggered migration does not need the extra PUT request. For migration within the same Amazon cloud region, such as from S3 to EBS, data transfer is free, and DELETE requests are also free.

2.4.3 Hints

The application can give additional hints to GRANDET for better prediction. A hint is an arbitrary set of key-value pairs. For example, a photo sharing application can provide the hint {user=alice, album=wedding} when storing an image file. The predictor will predict the workload of this file by considering files with similar hints, such as images uploaded by the same user in the same album. Hints are optional, and we have implemented two types of default hints if the application does not provide any hint. For the file system interface (see §2.5), the default hint is the directory hierarchy. For example,

if the photo sharing application stores a file at `alice/wedding/photo1.jpg`, the default hints would be `{hint1=alice, hint2=wedding}`. For the SDK interface (see §2.3.4), the default hints are the object’s key split by colons. We evaluate the effect of hints in §2.7.5.

2.5 File System Interface

Providing POSIX-like file system semantics is arguably the best way to support the widest range of legacy web applications seamlessly because it does not require modifications to their source code. Hence, GRANDET also implements a file system interface using FUSE. It can be directly mounted to the web application’s data directory.

The design of GRANDET’s file system interface follows our insight that most files are accessed wholly and sequentially by web applications, such as photo and video files. So, it is best to store each file as one object, as opposed to dividing files into blocks. Besides, web applications often need to rename files, such as moving a temporary file to its final directory. So, it is essential to support fast renaming, although S3 does not support renaming objects other than a copy followed by a delete. Last but not least, some web applications generate many intermediate files when doing backend processing, and remove them soon. So, it is desirable to skip putting these intermediate files to the backend storage.

Figure 2.7 shows the implementation of GRANDET’s file system interface. At the backend, it stores each file as a UUID-keyed object and puts the actual file name in its metadata. At the frontend, it maintains a *cache* of file contents on a RAM drive and keeps the *file structure* hierarchy and metadata (e.g., UUID, file size) in Redis. Therefore, renaming a file

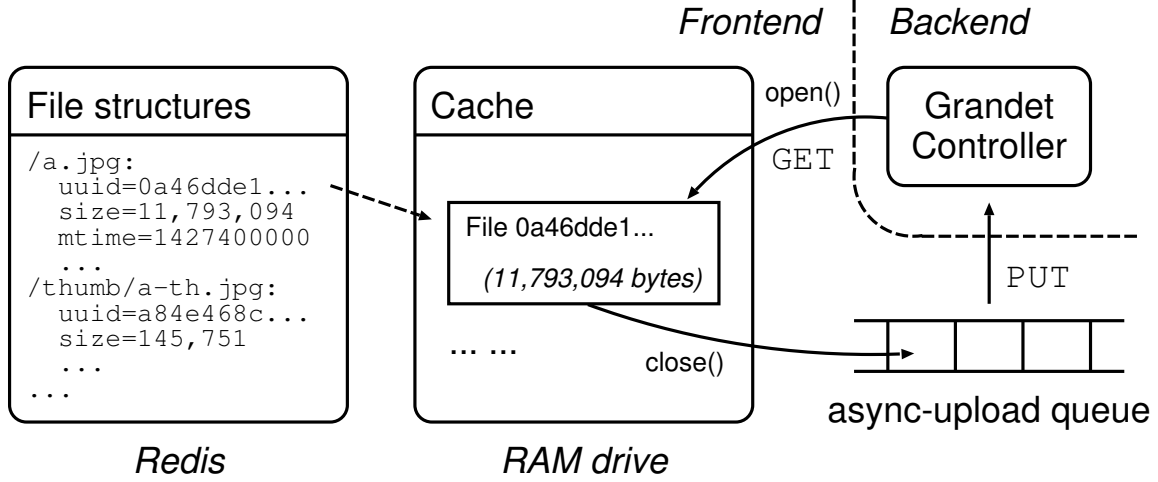


Figure 2.7: *Implementation of GRANDET's file system interface.* Solid arrows are the data flow. Dashed arrow shows the logical relationship between the file structure and its content.

only touches its metadata.

We next describe the file operations. On `creat()`, we create a file in the cache and pass the file descriptor to the application. On `open()`, we GET the file data from the backend storage if it does not exist in the cache, then open the cached file and return the file descriptor. For file manipulations such as read, write, and truncate, we pass them through to the corresponding file system operations of the cache. We also update our file structure for the new file size and modification time. On `close()`, if the file content has been modified, we append it into an *async-upload queue* so that the file will be PUT to the backend storage, and we block on `fsync()` until the PUT is completed.

Our implementation PUTs file contents to the backend storage asynchronously. It has two benefits. First, it skips intermediate temporary files if they are deleted before the actual PUT happens. Second, it allows an application to specify hints as extended attributes (`xattr`) efficiently after a file has been closed, which is useful when the creation of the

Component	LoC	Component	LoC	Component	LoC
S3 Actor	120	EBS Actor	236	Decider	230
Predictor	356	Controller	1401	Profiler	235
C++ SDK	159	PHP SDK	168	Python SDK	69
FS interface	1979	Console	349	Misc	1191
Total : 6493					

Table 2.4: *Lines of code of GRANDET’s components.*

file is beyond the application’s control, such as files generated externally. For example, the CumulusClips video sharing application executes `ffmpeg` to convert a video file to another format. It can set `xattr` of the converted file after that.

Since GRANDET’s backend makes the decision on the optimal storage location based on each file’s predicted usage pattern, the replacement algorithm on the cache is not critical. A simple LRU algorithm works well in practice.

2.6 Implementation and System Extensibility

We designed GRANDET as an extensible framework where each component, such as the storage services, the prediction algorithm, or the frontend SDK, can be easily replaced or extended. We implemented the GRANDET backend in C++14, the file system interface with `FUSE` [58], and key-value SDK in various languages. We modified `LIBAWS` [9] to communicate with Amazon Web Services. Table 2.4 shows the numbers of lines of GRANDET’s components. Metadata such as placement decisions are stored in Redis [116]. All components can be easily extended by plugging in a new subclass, or customized by changing a configuration file. This section describes some implementation details.


```

class Actor {
public:
    virtual void ~Actor()=default;

    // cloud storage operations
    virtual void put(const string& key, shared_ptr<Value> val)=0;
    virtual shared_ptr<Value> get(const string& key)=0;
    virtual void del(const string& key)=0;

    // updates pricing model, latency, availability, durability, etc.
    virtual void profile(shared_ptr<Profile> profile)=0;
};

```

Figure 2.8: *GRANDET's Actor class.*

2.6.1 Adding a storage service

GRANDET's Actor executes actions, such as PUT, GET, and DELETE, on the storage service. We implemented Actors for EBS (SSD and magnetic) and S3 (standard, reduced redundancy, and infrequent access). Supporting a new storage service just requires adding a new subclass of Actor and implementing its interface methods.

Figure 2.8 shows the interface of the Actor class. The `put()`, `get()`, and `del()` are cloud storage operations. The `profile()` method, when called by GRANDET's Profiler, updates the cloud service's Profile, which includes pricing model and service conditions such as latency, availability, durability, and consistency.

The Profiler is a cron job that periodically runs. When triggered, it calls every Actor's `profile()` method to update its profile. We implemented crawlers in our EBS and S3 Actors to fetch and parse the pricing information from the Amazon Web Services website. Profiles are stored as JSON files so that users can also manually configure the pricing model or service levels.

2.6.2 Adding a prediction algorithm

We implemented the prediction algorithm as described in §2.4. Plugging a new prediction algorithm into GRANDET just requires subclassing the Predictor class.

The Predictor has three listener functions, namely `notify_put()`, `notify_get()`, and `notify_del()`, which are called whenever there is a PUT, GET or DELETE request. The Predictor thus keeps track of the current workload. When making a decision, the decider calls the Predictor’s `predict_put()`, `predict_get()`, and `predict_lifetime()` functions to get the predicted future request frequency and expected lifetime.

2.6.3 Protocol and SDK

GRANDET’s frontend and backend communicate through Unix domain socket IPC, and all messages are serialized in Protocol Buffers [66]. GRANDET defines two types of protocol messages: Request and Response. A Request message is one of three types: PUT, GET, and DELETE. It also includes the key and value of the data object, the application’s requirements such as minimum durability and maximum latency, and optionally hints for workload prediction and other metadata. The Response message contains a status code, and optionally the data object’s value if it is the response to a GET request.

Therefore, the SDK for a programming language is just a wrapper over Protocol Buffer and socket programming. We have implemented the SDK for C++, PHP, and Python, with 70–170 lines of code each. We believe that supporting a new language would similarly require little programming effort.

2.6.4 Optimization

To further improve performance, we also implemented two optimizations to GRANDET’s basic design.

Shortcut for file access. When PUTting a file object that is already on disk, the request payload only includes the file name instead of the file content, and the GRANDET backend reads the file directly from disk. Therefore, it avoids sending the entire file from frontend to backend.

S3 authenticated URL. An application often GETs a data object from GRANDET only to send it verbatim to the user without any processing. For example, when a user clicks “download original image” on the Piwigo photo sharing application, Piwigo simply retrieves the data object for that original image and send it back to the user. Thus, if the data object is stored on S3, GRANDET incurs unnecessary overhead by acting as a proxy for the data transfer. In order to optimize for this scenario, the application can specify a special requirement in its GET request in the form of `{url=true, expire=600s}`; so the GRANDET backend sends the application a pre-authenticated URL for the S3 object with the specified expiration time (600s here). The application can thus redirect the user to download the image from the authenticated URL directly.

2.7 Evaluation

We evaluated GRANDET on four popular open source web applications: CumulusClips (video sharing) [39], Piwigo (photo sharing) [109], Elgg (social network) [46], and File-

Sender (file sharing) [54]. We modeled the usage data for each application according to the most popular website of its type, namely YouTube, Flickr, Facebook, and Dropbox. Section 2.7.1 details how we modeled the usage data. To make cost evaluations manageable, we scaled down the usage to 100 users in one month, while preserving real-world workload characteristics. We ran all experiments on EC2 m3.large instances with EBS and S3 in the US East region, using Ubuntu Linux 14.04 and Redis with per-second fsync.

Our experiments aim to answer four questions:

§2.7.2 Does GRANDET reduce cost?

§2.7.3 Is GRANDET fast?

§2.7.4 Is GRANDET scalable?

§2.7.5 Is GRANDET easy to use?

2.7.1 Workload modeling

We first describe our workload modeling for each app.

CumulusClips. We modeled the usage data of CumulusClips according to YouTube [146], where a billion users upload 300 hours of video per minute [147] and an average video has four minutes [98, 34]. So a user uploads 0.19 videos a month on average. Each month, an average user views 76 videos [34]. The average video size is 8MB [28]. On YouTube’s website, 20 recommendations appear with each video; so in our model, for each video viewed, 20 thumbnails are also viewed. A typical thumbnail has 400×300 pixels.

Piwigo. We modeled the usage data of Piwigo according to Flickr [56], where 87 million users upload 3.5 million images daily [80]. An average photo has 20 views [84]. On the Flickr website, an album shows 27 thumbnails. Large thumbnails have 1600×1000 pixels, and small ones have 640×400 pixels. A typical photo has 5120×3840 pixels.

Elgg. We modeled the usage data of Elgg according to Facebook [51], where 500 million users upload 120 million photos daily [17]. The average photo size is 60KB [17]. On Facebook’s website, a pop-up photo has 960×960 pixels and a thumbnail in the timeline has 300×300 pixels. Each day, 10 billion photos are viewed, and the ratio of views between thumbnails and original photos is 95% to 5% [17]. In our model, we distribute a user’s views among her friends following a Pareto distribution with $\alpha = 1.5$.

FileSender. We modeled the usage data of FileSender according to Dropbox [40], where 50 million users upload 500 million files daily [41]. We use the average size of files from file sharing servers, 153KB [135]. The popularity of the shared files follows a Zipf distribution with $\alpha = 0.4$ [137].

2.7.2 Cost savings

For the monetary cost, we evaluated each application’s end-to-end cost reduction, analyzed GRANDET’s operational cost, and tested its ability to handle dynamic workloads.

End-to-end cost savings

The overarching goal of GRANDET is to reduce cost used by web applications. Figure 2.9 shows a comparison of total storage costs of evaluated web applications with different

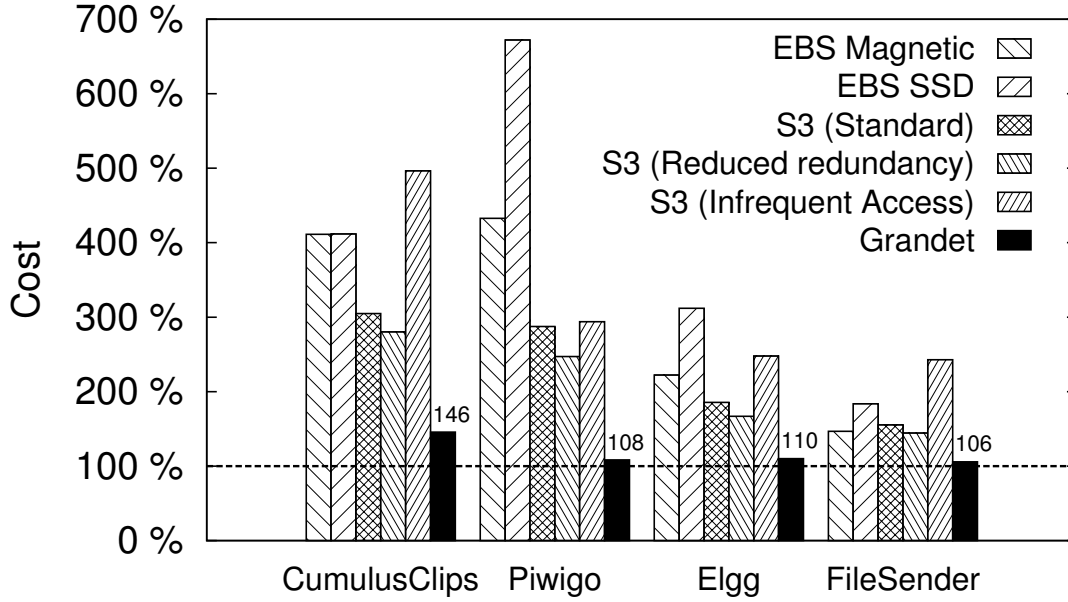


Figure 2.9: Comparing the cost of different backends and *GRANDET*. All costs are normalized to the optimal cost.

storage backends.² For each application, the first five bars are the cost of placing all objects into a single storage service. The last bar is the result of *GRANDET*'s dynamic placement. All numbers are normalized by the theoretically optimal placements, *i.e.*, each object is placed at the best storage if the entire workload was known beforehand (perfect prediction).

The results show that *GRANDET* always costs less than any single-storage option. It saves a geometric mean of 42.4% over the best single-storage setting. For example, *GRANDET* reduces Piwigo's cost by 56.2%. The reason is that Piwigo converts images into several resolutions, and images from different users and albums have distinct access patterns that are hard to be programmed statically but easy to be predicted dynamically by

²Storage costs in this dissertation were reported by *GRANDET* based on the precise storage space used and the number of requests recorded. We did not use Amazon's billing statement because it was too coarse-grained.

GRANDET.

Furthermore, for all but one applications, GRANDET's cost is within 10% of the optimal cost. For CumulusClips, although it costs 45.7% more than optimal, it is still 48.0% better than using any single storage backend.

It is worth noting that the cost saving ratio is independent of the number of users because the cost is proportional to the workload, which in turn is proportional to the number of users. Therefore, GRANDET is effective in reducing the cost for a broad spectrum of web applications.

Operational cost

To evaluate the operational cost that the GRANDET service itself incurs, we monitored its memory and CPU usage while running the Piwigo application. GRANDET only uses little memory; so we focus on CPU usage.

Assume that someone sets up a Piwigo instance to serve 100K users. Per our usage model (see Appendix 2.7.1), users would upload 120K photos and view 2.4M photos in one month. Meanwhile, 120K thumbnails would be generated, and they would be viewed 64.8M times. Thus, there would be a total of 2.52M large requests (95% read) and 64.92M small requests (99% read) per month, or 0.972 large requests and 25.1 small requests per second.

We evaluated GRANDET to see how many requests per second (RPS) it can handle per percent of CPU usage. In the worst case, GRANDET can handle 1.54 RPS per percent of CPU usage with large requests (1MB, 95% read), and 29.5 RPS per percent of CPU usage with small requests (4KB, 99% read). Plugging it into the above scenario, GRANDET would

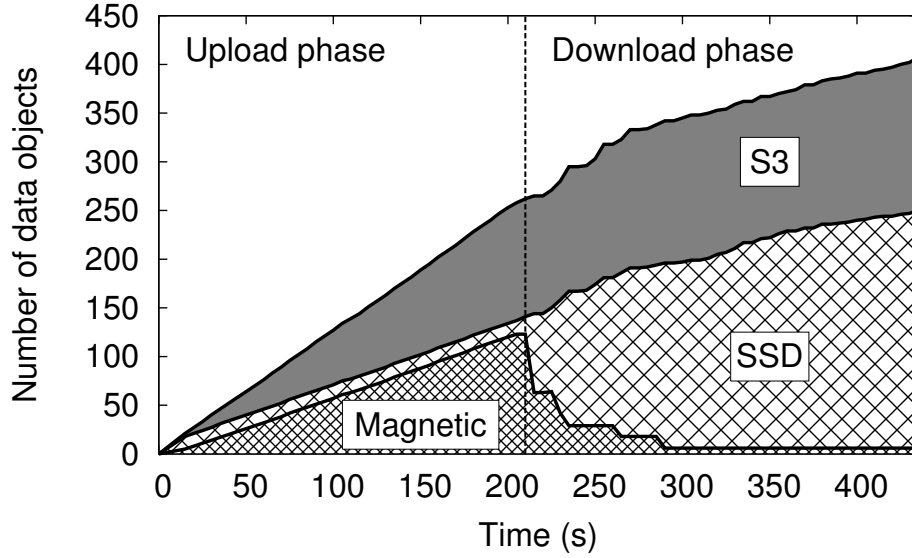


Figure 2.10: *Number of objects at each backend over time.*

consume 1.48% CPU to serve all requests. Since an EC2 m3.large instance costs \$38 per month and it has two cores, GRANDET only costs \$0.28 per month to serve 100K users in this case, negligible versus the storage cost it saves.

Handling dynamics

We evaluated how GRANDET reacts to workload changes by feeding it with an extreme case: we still send the requests according to real workload, but instead of sending requests like real users, we upload all data first and download afterward. Figure 2.10 shows how GRANDET behaves in this situation by showing the number of data objects stored in different storage backends over time. We used Piwigo in this experiment.

In the upload phase, GRANDET decides to put almost half of the objects on EBS magnetic, and most of the other objects on S3. Because there are not many requests, storing objects on S3 is cheap, and EBS SSD’s advantage of zero request cost does not help much.

On the other hand, EBS SSD’s storage cost is high; so few objects are put on SSD.

When the download phase comes, the predictor learns that some objects are frequently requested; so they are migrated out of EBS magnetic because its cost per request is higher than EBS SSD. Some objects are rarely requested, and some objects are large; so they are kept on S3. Some objects are migrated to EBS SSD, which has zero request cost and is ideal for objects with frequent access. The total number of objects is still increasing in this phase because some objects are lazily generated when they are accessed.

This experiment shows that GRANDET can adapt to workload changes over time, and the predictor is frequently using new information to optimize placement.

2.7.3 Performance

In order to understand GRANDET’s performance, we first measured over a microbenchmark and then evaluated each application’s end-to-end performance.

Microbenchmark

To evaluate GRANDET’s performance on basic operations, we evaluated each storage separately with a single client and two sizes of requests. Figure 2.11 shows the number of requests GRANDET can handle per second. Because the performance of GET requests is affected by the file system cache, we also measured the performance in the direct mode by specifying `O_DIRECT` on file system operations.

The performance of the EBS backends without cache matches the results of FIO [55], which measures the performance of the file system itself. Hence, GRANDET’s performance is limited by the hardware and underlying OS, and GRANDET itself incurs little overhead.

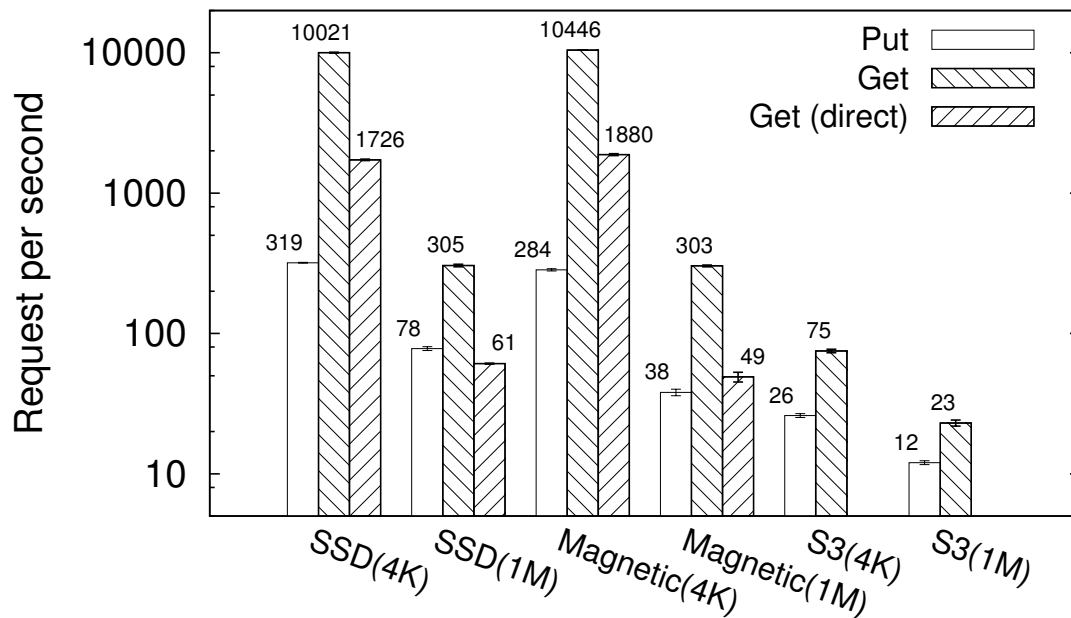


Figure 2.11: *Performance of GRANDET backend.* Evaluated with 4KB and 1MB requests. Error bar is standard deviation.

One interesting property of EBS disks is that they have different burst and sustained performance. For example, EBS SSD disks can reach burst throughput of 150MB/s, close to Amazon’s specification [7]. But after a few seconds, the throughput drops to about 60MB/s and keeps stable.

The cached GET requests of EBS backends are apparently served from the cache. The limiting factor here is the CPU speed. If the cloud provided better hardware, GRANDET would have better performance accordingly.

The results are low for S3 because S3 has a high latency for any request. Our profiler usually records the latency to be 20–30ms, and this latency limits the number of requests S3 can handle per second. Because S3 is not designed to handle frequent requests and it has a higher per-request price, it should not handle many requests.

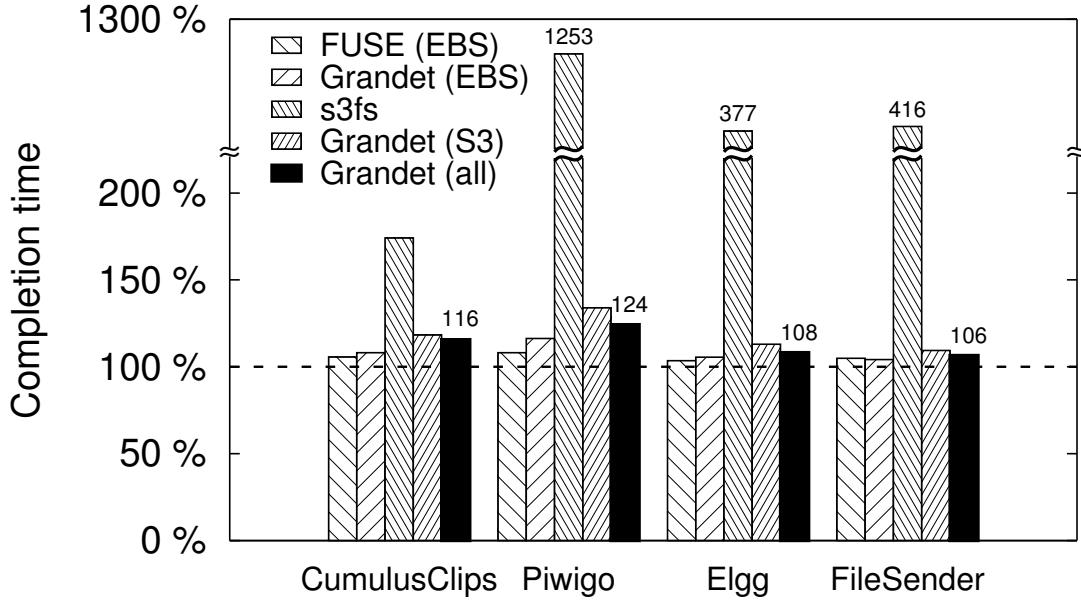


Figure 2.12: *GRANDET’s end-to-end performance*. Time is normalized to the baseline which uses EBS directly.

Latency. For all scenarios in the above experiment, we also measured the latency of each request. GRANDET always incurs less than 0.2ms latency, smaller than the standard deviation of latency for each case. Therefore, the GRANDET’s impact on latency is negligible.

End-to-end performance

We evaluated GRANDET’s end-to-end performance on the same four web applications. Because we use FUSE to implement the file system interface, we also evaluated the overhead incurred by FUSE itself. For comparison, we also ran the evaluation on the state-of-the-art S3-based file system s3fs [118]. Figure 2.12 shows the time used to complete the workload of each application and storage setting. We normalized all results to the baseline where all files were stored directly on EBS SSD. For the first bar in each cluster, a folder on the EBS SSD volume was mounted with the loopback FUSE file system to the appli-

cation's data folder. The second bar used GRANDET with only the EBS backend, so as to show GRANDET's overhead atop FUSE. The third bar used s3fs. As a comparison, the fourth bar used GRANDET with only the S3 backend. Finally, the last bar used GRANDET in the default setting with all backends.

GRANDET's overhead comes from several parts. The first part is incurred by FUSE, which averages to 5.5% (the first bar). The second part is incurred by GRANDET itself. Because using GRANDET with only the EBS backend has an average overhead of 8.5% (the second bar), the overhead incurred by GRANDET itself is less than 3%. The third part is incurred by the S3 backend, due to its higher latency than EBS. Mounting S3 as a file system with s3fs shows a prohibitive average overhead of 330% due to synchronous uploads and limited metadata cache (the third bar), whereas GRANDET's average overhead using only the S3 backend is 18.3% (the fourth bar). Overall, GRANDET incurs a geometric mean of 13.3% overhead (the last bar), which can be offset by the cost it saves.

2.7.4 Scalability

To evaluate GRANDET's scalability, we measured over both a microbenchmark and a web application.

Microbenchmark

In order to see whether GRANDET can scale up, we evaluated GRANDET with a variable number of concurrent threads and variable request sizes on variable storages. The results are similar, and for brevity, we show a typical one: S3 with request size of 4KB. Figure 2.13 shows the performance of the server when the number of concurrent clients increases.

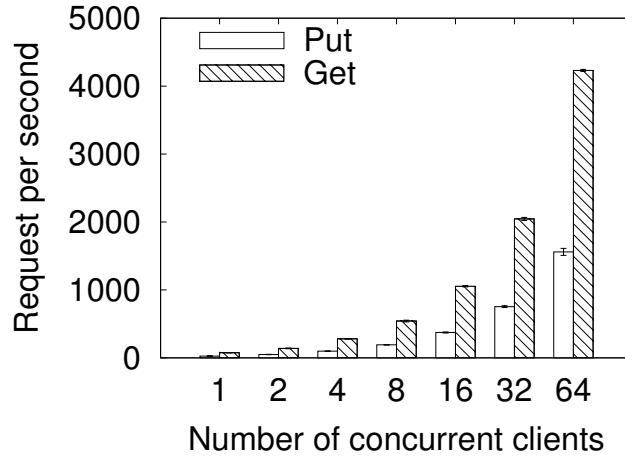


Figure 2.13: *Scalability of GRANDET when using single S3 storage.* Evaluated with requests of 4KB in size. The error bar is the standard deviation.

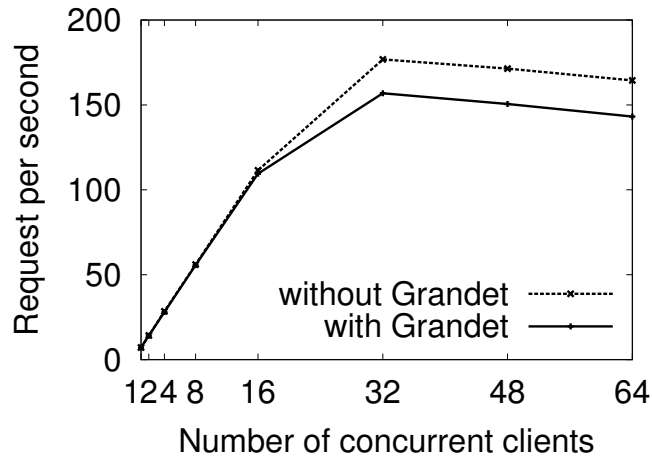


Figure 2.14: *End-to-end scalability.* Evaluated on the FileSender application with real workload.

The number of requests the server can handle per second increases almost linearly. It implies that the number of requests one client can achieve is limited by the latency of the S3 service, and the server scales well with the number of clients.

End-to-end scalability

We evaluated the end-to-end scalability of GRANDET by measuring the number of end-to-end requests the system can handle when the number of clients increases. The requests go all the way through Nginx, PHP, FUSE and the GRANDET backend. We chose the most scalable application—FileSender—among all the applications we studied, so that if there were any scalability issues with our system, it would be revealed by the experiment. The FileSender application is the most scalable application because of its simplicity: it does not perform any operations on the files, but just lets other users download them.

Figure 2.14 shows the requests per second with a variable number of concurrent clients. The results show that GRANDET scales as well as FileSender. Regardless of whether using GRANDET, FileSender does not scale past 32 concurrent clients, which is due to limited resource in the EC2 m3.large instance, not GRANDET's limitation.

2.7.5 Usability

GRANDET can run a web application unmodified and automatically save cost. We have also tested and confirmed that three of today's most popular web applications—MediaWiki, Wordpress, and Joomla—work seamlessly with GRANDET without any source code modification.

To further reduce cost, application developers can add hints to data objects. In all our evaluations, we did not add hints to CumulusClips or Elgg but added one hint to each of Piwigo and FileSender. We found that compared with using the default predictor, hints helped reduce cost by 9.3% for Piwigo and 9.4% for FileSender.

2.8 Discussion

We now discuss some design implications of GRANDET.

Persistence over server crash. If the GRANDET server crashes, all data objects that have been PUT onto EBS or S3 will persist. Metadata (*e.g.*, placement decisions) rely on the persistence of Redis, which can be configured as AOF (log-based), RDB (snapshot-based; metadata lost since the last snapshot can be rebuilt from the cloud, similar to `fsck`), or both. For the file system interface, the local cache may not persist, which would affect data objects in the async-upload queue that have not yet been PUT to the backend storage. GRANDET provides the same semantics as a file system by blocking on `fsync()` until the PUT is complete.

S3 consistency. S3 provides read-after-write consistency for new objects and eventual consistency for overwrites. There are two ways to work around it. First, the application can specify in each object's requirement to avoid S3. Second, GRANDET can use versioning in S3 placement decisions so that each PUT operates on a new object.

Data replication across cloud regions. Because EBS volumes can only be accessed within a cloud region, GRANDET's server must reside in the same cloud region as all EBS volumes. However, since GRANDET exposes a general key-value object store interface, it can be easily extended to multiple cloud regions by overlaying existing geo-replication solutions atop GRANDET.

Migration granularity. GRANDET migrates data lazily on a per-object basis; so data objects that are not accessed would not be migrated, even if better storage choices were

available. One way to solve it is to have a thread periodically scan through all objects to find migration possibilities. In practice, the changes of workload on data objects are gradual, so that a cold object would already be migrated before its access drops to absolute zero.

EBS elasticity. Adjusting EBS volume size takes minutes to finish. GRANDET can leverage existing orthogonal strategies (*e.g.*, [115]), or rely on application developers for allocating EBS volumes. Amazon’s recently-announced elastic file system (EFS) is fully elastic and does not have this issue. GRANDET can support it by just adding an Actor for it.

Metadata overhead. GRANDET’s metadata is on the order of tens of bytes per object, comparable to a regular file system. It is negligible for data files in typical web applications, but may not be suitable for storing a lot of tiny objects. Cloud database services, such as Amazon DynamoDB, complements GRANDET for database or tiny-object storage.

2.9 Related Work

GRANDET builds upon prior work that we now describe.

S3 lifecycle. Amazon has rudimentary support for moving S3 objects to the infrequent-access option or Glacier. However, such transitions are one-way and limited to S3, and developers must set rules manually. GRANDET supports automatic transitions across all storage options.

Cloud economics. Some recent work studies the economics of cloud computing. Much of the work is focused on reducing the cost of computing, not storage. For example, Tak

et al. [130] discusses the cost factors for several cloud-based application deployment options, and Conductor [142] optimizes cloud service choices for MapReduces computations. Other work touches upon storage. CloudCmp [85] provides a microbenchmark suite for measuring the cost and performance of different cloud service providers. Developers can then inspect the benchmark results and pick a provider for their application. GRANDET may leverage this microbenchmark suite in its profiler implementation.

Cloud-backed file systems. Several systems provide a file system interface atop a blob storage such as S3. Open source projects, such as s3fs [118], s3ql [119], and goofys [62], can mount an S3 bucket as a local file system. The BlueSky network file system [134] employs a log-structured design on the cloud storage. SCFS [20] enables sharing for cloud-backed file systems. These systems assume general file system workloads, and the main challenges they tackle are performance issues, such as how to support random writes atop a blob storage that does not support partial updates. Unlike GRANDET, these systems do not exploit the characteristics of files used by web applications or reduce monetary cost.

Multi-tier storage systems. Multi-tier storage systems are widely used today, such as FAST [47], Easy Tier [44], 3PAR [72], and some recent work [68, 127, 151, 138]. These systems migrate data among traditional storage, while GRANDET works with cloud storage services.

Cloud-of-clouds. Several pieces of work propose the idea of storing data across multiple clouds. Some do so to replicate the same data multiple times for fault tolerance. For example, RACS [1] applies the RAID technology to cloud systems. DepSky [19] uses

multiple services for dependability and security. MetaStorage [18] uses multiple services to manage consistency-latency trade-offs. NCCloud [73] applies network coding to cloud storage for fault tolerance. These systems aim to increase durability and availability, not to reduce cost. In fact, by storing more copies of data, they increase monetary cost, which GRANDET can help reduce.

Other pieces of work, including FCFS [115], iCostale [2], Scalia [104], and SPANStore [144], store data across clouds for reducing cost, a goal similar to GRANDET's. FCFS only has simulations showing potential savings of storing objects across different cloud services, which serve as an excellent motivation for GRANDET. ICostale and Scalia also do simulations only, and they consider only blob storage which cannot support many popular web applications. To the best of our knowledge, none of FCFS, iCostale, or Scalia provide a system that developers can use. SPANStore also considers only blob storage; so it also requires modifications to many web applications. In addition, its coarse-grained placement decisions only consider geographical locations. In contrast to these systems, GRANDET makes fine-grained predictions and decisions based on each data object's own characteristics and access pattern, and it works seamlessly with today's web applications without modifications.

2.10 Summary

This chapter presented GRANDET, an extensible storage system that significantly reduces storage costs for web applications deployed in the cloud. It unifies multiple heterogeneous stores by placing each data object at the most economical store and provides both a file

system interface and a key-value SDK. Evaluation on a diverse set of popular open-source web applications shows that it reduces costs by an average of 42.4%, and it is fast, scalable, and easy to use. Its source code is at <http://columbia.github.io/grandet>.

Chapter 3

Secure Deduplication of General Computations

The world's data has been fast exploding for many years. It is estimated that in 2011 alone, 1.8 zettabytes of data were created, and the overall data will grow by 50× by 2020 [91]. This massive amount of data comes in greatly varying forms, ranging from personal photos and videos, to office documents and web pages, to source files, binary programs, and virtual machine images, and to data collected from user clicks or physical sensors.

Meanwhile, the storage of this data has become highly concentrated. It is common practice for enterprises to store data on centralized, powerful storage servers for ease of management [140]. The cloud computing paradigm has migrated data into the cloud so that the computations can be closer to the data. For instance, several organizations have put 56 public data sets totaling 761.2TB onto Amazon Web Services [113]. Even consumers are beginning to aggregate their personal data into the cloud for convenience. For instance, Google, Dropbox, Amazon, and Microsoft all provide the option for users to automatically upload pictures and videos shot using their mobile devices. Facebook stores over 260 billion personal photos [17].

This highly concentrated, massive data poses challenges for storage provisioning and management. Fortunately, prior work has shown that a significant portion of the data is redundant [92] and that *data deduplication* can hugely reduce the storage needed to hold

the data and simplify management [43]. For instance, *file deduplication* detects when multiple files have the same data and stores the unique data only once [22]. This scheme is particularly useful when the same file is copied, such as when a user makes a copy of her friend’s shared video on Dropbox. *Block deduplication* breaks files down to variable [99, 90] or fixed [150] size blocks and stores each unique block of data once. This scheme is particularly useful for files that are similar but not exactly identical, such as different versions of a document and virtual machine images built from the same OS family. These deduplication schemes have been long prevalent in enterprise storage servers [43]. With the trend of moving consumer data into the cloud, these schemes have also become popular among cloud storage providers such as Dropbox [126].

Not only can data be redundant, the computations on top of the data can also be redundant. For instance, a user may scan her Dropbox files for viruses, while another user runs the same virus scanner on a similar set of files. Different users may be doing the same computations on the public data sets in AWS, such as building an inverted index for the web pages in CommonCrawl [32]. Given the same input data, the same deterministic computation always produces the same result. Thus, if the computation is slow, it is typically more efficient to memoize [93] and reuse the result than redoing the computation. We term this technique *computation deduplication*.

Several prior systems deduplicate computations (e.g., [26, 69]). However, three main challenges prevent these systems from effectively deduplicating computations in today’s cloud or enterprise environments:

First, how can we deduplicate computations done by *mutually distrusting* users? Storage providers such as Dropbox aggregate data from many users who do not necessarily

trust each other. Even in an enterprise setting, users frequently have different data access permissions. One naïve approach is to memoize computation results in a cache every user can read or write, but this approach provides neither integrity or security. A malicious user can easily poison the cache, by for instance marking files that contain viruses safe. She can also read results in the cache even though she has no permission to access the actual data in the results. Although this challenge may be solved with information flow tracking or access control systems, these systems are known to be difficult to configure and use.

Second, how can we deduplicate *general* computations? Prior systems deduplicate computations purely at the system level, assuming no cooperation from application developers. As a result, they handle only specific computations. For instance, ccache [26] deduplicates only the compilations of C/C++ programs, and Nectar [69] deduplicates the computations of programs written only in DryadLINQ [149], a specially designed language for large scale data-parallel workloads. However, the computations that users want to do on their data can be extremely rich, and it is unrealistic to require storage providers to understand all of them. For instance, while it may be feasible for Amazon to run some basic virus scanning software on the files it hosts, it is impossible for Amazon to understand every advanced virus scanner, every compression tool, and every image/video manipulation utility users want to run on their data.

Third, how can we effectively deduplicate computations on top of *deduplicated data*? Prior systems rely on custom methods to detect that data is redundant. For instance, ccache computes a hash of a preprocessed C/C++ source file and uses this hash to search its compilation cache. These methods incur unnecessary overhead when the data is dedu-

plicated because the underlying storage system already knows what data is redundant.

This chapter presents UNIC,¹ a system that securely deduplicates general computations. It exports a cache service that allows applications running on behalf of mutually distrusting users on local or remote hosts to memoize and reuse computation results. Key in UNIC are three new ideas:

First, through a novel use of code attestation, a classic primitive to attest what code is running to a (remote) party [125, 124], UNIC achieves both integrity and secrecy. To insert or query the result cache that UNIC maintains, UNIC generates a secure, non-forgable key that attests to both the application code and the input data. This key strongly isolates applications from each other in the result cache. For instance, if a malicious user modifies the code of a virus scanner in attempt to poison the cached results of this virus scanner, the attempt would fail because the modified code leads to a different key. In addition, since this key is not forgeable, a malicious user cannot query UNIC’s cache without already knowing the application code and the input. Since the user knows the code and input already, she can already compute the result by herself.

Second, UNIC provides a simple yet expressive API that enables applications to deduplicate their own rich computations. From a high level, this API supports an application to (1) insert *input* \rightarrow *result* to the result cache UNIC maintains, and (2) query the cache with *input* and get back the cached *result* if any. This application-level computation deduplication design is much more general and flexible than prior system-level designs.

Third, UNIC explores a cross-layer design that allows the underlying storage system

¹We name our system UNIC (pronounced “unique”) because it is conceptually similar to the Unix *uniq* utility applied to computations.

to expose data deduplication information to the applications for speed. Applications thus do not need to re-detect whether the input data is redundant. For instance, suppose two files A and B are identical so the filesystem deduplicates them, and UNIC exposes this data deduplication information to the applications. After a virus scanner scans file A, it can immediately skip file B without reading any data from B, significantly increasing its scanning speed.

Our implementation of UNIC stores cached results in Redis, a fast, scalable, replicated key-value store [116]. UNIC implements code attestation in a dynamically loadable Linux kernel module and considers the kernel to be trusted. It implements the computation deduplication API as a library, which applications link with. UNIC leverages ZFS [150], a file system that supports both file and block deduplication, to detect when data is deduplicated on behalf of the applications running with UNIC.

Evaluation of UNIC on four popular open-source applications shows that (1) it is easy to use (to support each application, we needed to change fewer than 1% lines of source code); (2) it is fast (it sped up applications by up to 21.4×); and (3) it incurs little storage overhead (it needed only 3.45% additional storage to cache the results).

The remainder of this chapter is organized as follows. The next section discusses the security model and UNIC’s design. §3.2 describes UNIC’s API and usage. §3.3 presents how UNIC leverages deduplicated data. §3.4 describes the implementation. §3.5 shows evaluation results. §3.6 discusses UNIC’s security implications, §3.7 describes related work, and §3.8 summarizes this chapter.

3.1 Security Model and Design

We begin with UNIC’s assumptions, threat model, and the design of UNIC’s protocol.

3.1.1 Assumptions and Non-assumptions

First, UNIC relies on a code attestation mechanism for integrity and secrecy of the cached results. It leverages this mechanism to bind a result to the code and input data that together produce the result. This mechanism can be implemented in multiple ways with different security strengths. For instance, UNIC could use TPM and isolation technologies such as Intel TXT [76] to realize code attestation, but doing so would incur both deployment and runtime overhead, negating our goal of being easy to use and fast. Therefore, for practical reasons, UNIC assumes that the OS is trusted and provides a function to attest the application code, and that the user does not have superuser privileges to interfere with that mechanism. This assumption matches well with many of today’s mobile devices that run Chrome OS [64], iOS, and Android.

Second, UNIC assumes correct application code. For instance, when using UNIC, an application developer should use UNIC’s API correctly. She should only memoize computations with deterministic results. UNIC also assumes that the application is free of vulnerabilities such as buffer overflows. We note that this assumption is common to almost all prior code attestation work.

Third, UNIC assumes that its underlying storage system provides reasonable security guarantees. To reuse results across sessions, UNIC persists them in an underlying storage system such as a file system. UNIC assumes that this storage system is properly config-

ured such that an attacker cannot access the data stored without going through UNIC. This guarantee and UNIC’s security mechanisms described in §3.1.3 together ensure the integrity and secrecy of its cache of computation results.

3.1.2 Threats

UNIC enables deduplicating computation among mutually distrusting users. Two attacks are particularly serious for UNIC: *cache poisoning* attacks UNIC’s integrity, and *query forging* attacks UNIC’s secrecy.

Cache poisoning. A malicious user may write a new application or modify an existing application in an attempt to poison the result cache. Her application may attempt to insert or overwrite entries belonging to a legitimate application. UNIC prevents this attack by isolating applications in the result cache: it guarantees that the cached data for one application can never be accessed by another application. Specifically, UNIC securely binds the computation code and the input data to the computation result leveraging a code attestation mechanism.

Query forging. A malicious user may write a new application or modify an existing application in attempt to query entries in the result cache that she cannot access, and gain information. UNIC prevents this attack again by isolating applications. When an application queries the cache, UNIC generates a search key that attests to both the code and the input data that generate the query. This key is unique to each application. One application thus cannot query entries of another application.

Several other attacks are possible, some of which can be prevented using simple mech-

anisms such as rate-limiting queries sent to UNIC. We briefly describe how they can be prevented in §3.6, and leave the implementation for future work.

3.1.3 Design

UNIC novelly leverages code attestation to cryptographically bind the *result* with the *code* and the *input* that produced the *result*, preventing cache poisoning and query forging attacks.

UNIC assumes a trusted OS that securely computes SHA-1 hash and HMAC. A secret key K is shared among trusted OSes. (Existing work [125] details how to distribute this key. We use symmetric key for efficiency; however asymmetric key works, too.) An attacker cannot forge $\text{HMAC}(\text{data}, K)$ without knowing K .

UNIC leverages code attestation to bind *result* to *code* and *input* that produced *result*. Specifically, it uses code attestation to compute two things:

(1) $\text{result} = \text{code}(\text{input})$

// Run *code* on *input* to compute *result*.

(2) $\text{sig} = \text{HMAC}(\text{hash}(\text{code}) || \text{hash}(\text{input}) || \text{result}, K)$

// Bind *code*, *input*, and *result*. We use $||$ as the concatenation operator.

The assumptions on trusted OS, unprivileged user, and correct application code together guarantee that *result* is the correct result of running *code* on *input*. This code attestation mechanism further guarantees that (a) *sig* cryptographically attests that *result* is indeed produced by running *code* on *input*, which anyone with access to *code*, *input*, *result*, and K can verify; and (b) *sig* cannot be forged.

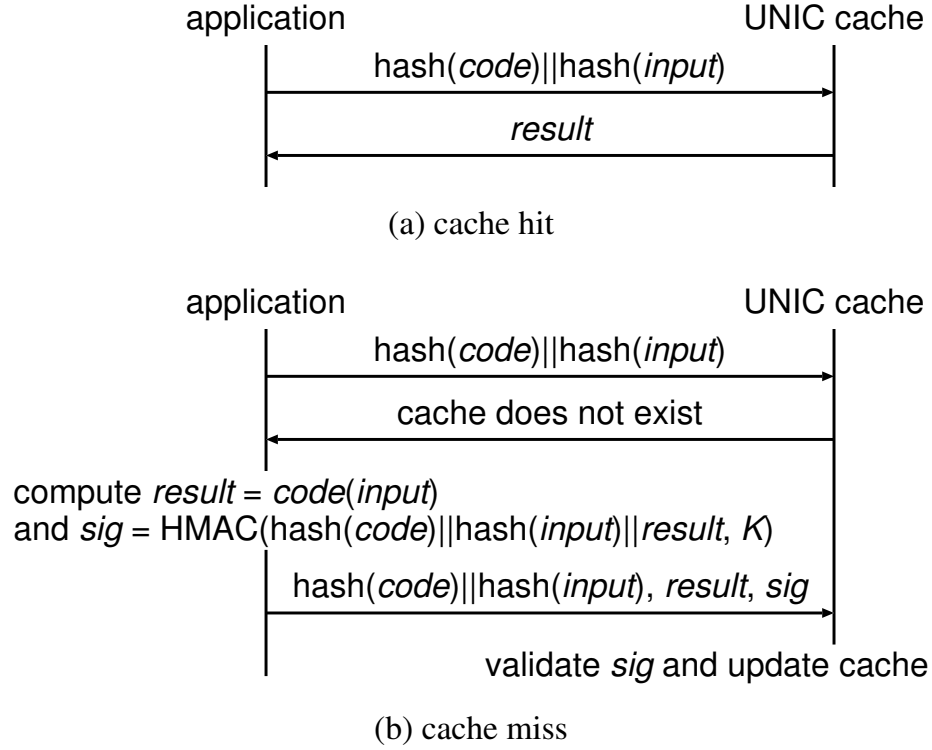


Figure 3.1: *UNIC protocol*. We use $||$ as the concatenation operator.

UNIC protocol. The UNIC cache is a mapping of

$$\text{hash}(\text{code}) || \text{hash}(\text{input}) \rightarrow \text{result}$$

Since the hash function is collision resistant, the cache space for different computations are isolated.

When an application wants to compute $\text{code}(\text{input})$, it sends $\text{hash}(\text{code}) || \text{hash}(\text{input})$ to the UNIC cache. If cache exists (Figure 3.1(a)), UNIC sends back result . If cache does not exist (Figure 3.1(b)), the application computes both result and sig , and sends $\text{hash}(\text{code}) || \text{hash}(\text{input}), \text{result}, \text{sig}$ to the UNIC cache. The UNIC cache validates that sig is indeed $\text{HMAC}(\text{hash}(\text{code}) || \text{hash}(\text{input}) || \text{result}, K)$, and updates the cache.

3.1.4 Security Analysis

The design of UNIC prevents cache poisoning as follows. Suppose an attacker replaces *result* with *bad_result* when inserting into UNIC. Because of code attestation, she cannot forge *sig*, so UNIC cannot validate *sig*. Suppose she modifies *code* into *bad_code* and computes *bad_result* to poison the cache. Because UNIC validates *sig*, she can only insert

$$\text{hash}(\text{bad_code}) || \text{hash}(\text{input}) \rightarrow \text{bad_result}$$

which cannot affect the cache entry of $\text{hash}(\text{code}) || \text{hash}(\text{input})$. To avoid a malicious client from polluting the cache space, UNIC can employ a quota mechanism to limit the cache space for each client application.

This design also prevents an attacker from forging a query to steal *result*. To query cache, she must send $\text{hash}(\text{code}) || \text{hash}(\text{input})$, so she must already have *code* and *input* because otherwise she would not be able to compute the hashes. Once an attacker has *code* and *input*, she can already compute *result* simply by running *code* on *input* herself. Thus, she cannot gain additional information with this query other than whether there is a result in the cache. §3.6 further discusses its implications.

3.2 UNIC API and Usage

UNIC provides a simple yet expressive API for applications to deduplicate their own rich computations. We first motivate our API design through an example, and then formally describe its interface.

```
1: void simple_virus_scanner(file, options) {  
2:   buffer = read(file);  
3:   result = scan_signature(buffer, options);  
4:   print(result);  
5: }
```

Figure 3.2: *A simple virus scanning application.*

3.2.1 Example

We motivate the design of UNIC API through a step-by-step example showing how a simple virus scanning application could use memoization to deduplicate computation. Conceptually, the application works like Figure 3.2. It reads the file content into a buffer, executes virus scanning algorithm on the buffer, and outputs the result.

In this piece of code, line 2 reads the file content from disk, potentially a time-consuming I/O operation. Line 3 performs some CPU-bound virus signature matching algorithm, potentially another time-consuming operation. Line 4 prints the result, which is relatively fast because the length of the scanning result (e.g., “no virus found”) is much smaller than the original file content. Therefore, we want to improve the performance on lines 2 and 3.

Memoizing Computations. We first examine how to use memoization to avoid duplicate computation on line 3. Since `scan_signature()` is a deterministic function over the input buffer and the signature-scanning options, if we could memoize the result the first time we perform the computation, we would be able to safely reuse the result later on the same input. To do so, we modify the application into Figure 3.3, using three functions that UNIC provides: `exists()`, `get()`, and `put()`. It first checks if the computation for

```

1 : void simple_virus_scanner(file, options) {
2 :   buffer = read(file);
3 :   if (exists(scan_signature, buffer, options)) {
4 :     result = get(scan_signature, buffer, options);
5 :   } else {
6 :     result = scan_signature(buffer, options);
7 :     put(scan_signature, buffer, options, result);
8 :   }
9 :   print(result);
10: }

```

Figure 3.3: *First step: memoize the computation result.*

the given buffer and options exists in the result cache (line 3). If so, it simply gets the memoized result (line 4). Otherwise, it performs the computation as before (line 6) and then puts the result into the cache (line 7).

As discussed in §3.1.3, the cache is not merely a mapping from the input to the result, but binds the computation code together with them. UNIC internally computes a non-forgeable authentication code that guarantees that the result (`result`) is indeed generated by the computation code (`scan_signature()`) over the input (`buffer` and `options`). The result cache is updated only if it can verify this authentication code.

Reducing I/O Operations. Memoizing the computation is good, but it would be better if we could also eliminate the need of reading the file content on line 2. This is not trivial because if we did not read the file in the first place, we would never know if the signature scanning is performed on the same content. Fortunately, it is possible if the file is stored on a deduplication-enabled storage.

A deduplication-enabled filesystem, such as ZFS [150], stores all files with the same content as a single copy. It does so by identifying the file content using a cryptographically

```

1 : void simple_virus_scanner(file, options) {
2 :   hash = get_file_hash(file);
3 :   if (exists(scan_signature, hash, options)) {
4 :     result = get(scan_signature, hash, options);
5 :   } else {
6 :     buffer = read(file);
7 :     result = scan_signature(buffer, options);
8 :     put(scan_signature, hash, options, result);
9 :   }
10:   print(result);
11: }

```

Figure 3.4: *Final version: use filesystem metadata to further reduce I/O operations.*

collision-resistant hash (e.g., SHA-256), and mapping all files with the same content to the same hash. These hashes are stored on the filesystem metadata, separate from the actual file content. Therefore, it creates a perfect opportunity for our application to tell if the file contents are the same without actually reading them.

Figure 3.4 shows the final version of the application. Instead of reading the file content up front, it now gets the unique hash of the file directly from the filesystem metadata using UNIC’s `get_file_hash()` function (line 2), and uses the hash to identify the memoization (lines 3, 4, and 8). Since getting the hash is much faster than reading the whole file, we have further avoided the slow I/O operation when reusing a previously cached computation.

In practice, when using UNIC, the application developer does not need to worry whether the storage has deduplication enabled or not — she should always follow the final version in Figure 3.4 and use hash to identify the memoization. This is because UNIC *transparently* leverages storage deduplication information. Where such information is absent, UNIC computes and caches the hash by itself. This process is detailed in §3.3.

3.2.2 The API

The previous example illustrates the usage of the UNIC API which we now formally describe. It wraps OS- and filesystem-specific details by exporting the following functions:

- `init()` initializes UNIC.
- `get_file_hash(file)` returns the hash of a file, where `file` can be the name of a file, a file descriptor, or an inode number. If the underlying filesystem has deduplication enabled (*e.g.*, ZFS), it gets the hash of the file from the filesystem metadata without reading the file content. Otherwise, it computes the hash from the file content using `libcrypto`.
- `get_block_hash(file, block)` is similar as above, but returns the hash of a block of a file, where `block` specifies the block number. This is particularly useful if the application's computation is based on blocks, such as a `bzip2` compression. The application should decide whether to use `get_file_hash()` or `get_block_hash()` based on its own logic, which is discussed in §3.3.
- `exists(computation, hash, id)` checks if a given computation and input exists in the result cache. The parameter `hash` is the hash of input data. The parameter `id` is an optional string identifier defined by the application, used for differentiating multiple computations performed on the same input. For example, the virus scanning application may let `id` be the signature-scanning options.
- `get(computation, hash, id)` gets the result of a given computation and input from the result cache.
- `put(computation, hash, id, result, ttl)` puts an entry of computation, in-

put, and result into the result cache. An optional `t1` specifies its time-to-live in seconds, and the result cache automatically deletes the entry upon expiration.

3.3 Leveraging Storage Deduplication

UNIC explores a cross-layer design allowing underlying storage system to expose data deduplication information to the applications.

Typically, a deduplication-enabled filesystem maintains the hash of each file as its metadata. Since UNIC also uses hash to identify the memoization input, it is both convenient and efficient to leverage such filesystem metadata. Therefore, when an application needs to get a hash, UNIC automatically detects the underlying storage system type, and returns the hash directly from the metadata if the filesystem has enabled deduplication. If not, UNIC reads the file content and computes the hash itself. In this way, UNIC provides a consolidated interface for both scenarios, making the storage system details transparent to the applications.

Furthermore, the application does not need to know whether the underlying storage system is file-level or block-level deduplicated. It should decide whether to use `get_file_hash()` or `get_block_hash()` solely based on the application's own logic. Generally, if the application's computation works with the file on a block-by-block basis, such as the bzip2 compression algorithm, it should use `get_block_hash()`. Otherwise, if the application's computation uses the file as a whole or randomly accesses the file, such as an anti-virus program, it should use `get_file_hash()`.

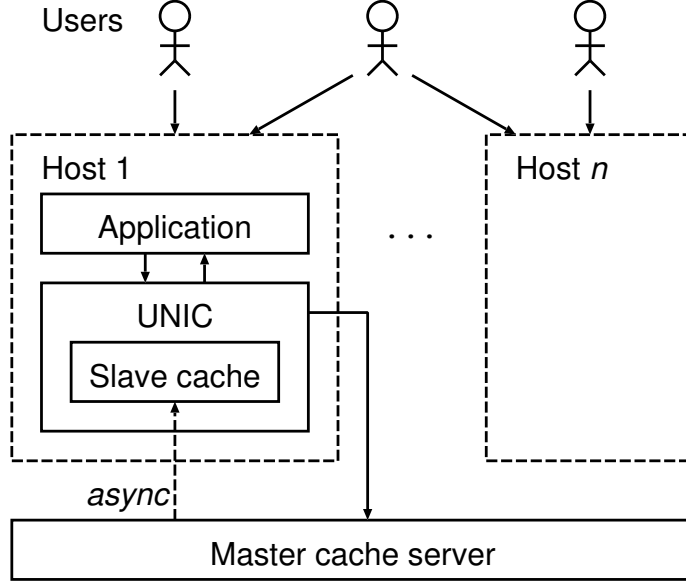


Figure 3.5: *UNIC architecture*. Additional hosts each have the same architecture as Host 1, and are omitted here due to limited space.

3.4 Implementation

We now describe UNIC’s components and implementation details.

3.4.1 UNIC Components

Figure 3.5 shows the architecture of UNIC. It is deployed on a network of multiple hosts. Each user can log into multiple hosts, and each host can have many users logged in. Because of UNIC’s security design (§3.1), different users do not need to mutually trust each other.

The UNIC module on each host handles application’s memoization requests. Since memoization works best when the reuses of computations are frequent, reading data from the result cache should be more common than writing data to it. In light of this, we design UNIC to make read operations as fast as possible. A trusted master cache server handles all

write operations. It can be either standalone or co-located with the enterprise's storage (e.g., NFS) server. Each host has an optional read-only slave cache, which periodically syncs from the master cache server. If the slave cache is present, all read operations happen locally. For security, all network communications are encrypted with SSL/TLS. To reduce the handshake latency, the UNIC module on each host establishes a connection with the master cache server when the host boots up, and keeps the connection alive.

Because data updates on the slave caches happen asynchronously, it is possible that a host does not have the latest cached results. However, we point out that memoized computations are deterministic (§3.1.1), therefore the consistency on the slave caches should not affect the integrity of computations. The only contingency would be that an application may not be able to leverage recently cached results but have to compute on its own.

UNIC inserts a kernel module into the Linux kernel as a virtual device for computing $\text{hash}(\text{code})$ and sig . It represents code by the image of the executable process, with all libraries statically linked. The secret key K is inaccessible to the user space. The user-space application talks to the kernel module via `ioctl`. For improved performance, the kernel module internally caches $\text{hash}(\text{code})$ for each caller.

UNIC uses a modified Redis key-value store [116] as the result cache. It modifies Redis to support UNIC's protocol (§3.1.3), and removes nonessential functions (such as `KEYS` which can list all cache entries) from Redis for security. Therefore, users cannot access the result cache except through UNIC.

3.4.2 Opportunistic Memoization

When using UNIC, the application developer needs to judge the best opportunity to use memoization because of two reasons. First, memoizing an already-fast computation may not justify the overhead of accessing the result cache. Second, abusing memoization for low-redundancy computations could result in exceeded overhead for entries that are never reused later. However, making the optimal decision at compile time is usually hard because input data cannot be predicted. Therefore, UNIC provides an optimization to opportunistically enable memoization only when the computation is slow and its reuse happens to be frequent at runtime.

To do so, UNIC internally has a model of $T_{put}(\text{result_size})$ and $T_{get}(\text{result_size})$, meaning how long it would take to put and get a certain size of result, respectively. This model is independent of the actual content of the result, and it can be learned from a microbenchmark upon the installation of UNIC (see §3.5.2 for our evaluation). UNIC also maintains an accumulator t_{save} for each computation, initialized to 0, for the total time that could have been saved for the future.

UNIC further provides two functions for an application to mark the boundary of a computation. An application calls `begin()` to indicate that a computation starts, and UNIC records the current timestamp as t_{begin} . An application calls `end()` to indicate that the computation has finished, and UNIC records the current timestamp as t_{end} . When `put()` is called, UNIC does not put the data into the result cache immediately, but updates t_{save} to be

$$t_{save} = t_{save} + t_{end} - t_{begin} - T_{get}(\text{result_size})$$

Therefore, the slower and the more frequent a computation is, the larger t_{save} becomes. UNIC only performs the `put()` operation when t_{save} is greater than $T_{put}(\text{result_size})$, *i.e.*, the time that could have been saved from a computation is greater than the time that would be spent for memoizing the computation. In the case that $t_{save} < T_{put}(\text{result_size})$, UNIC ignores the `put()` request, and simply updates t_{save} .

3.5 Evaluation

We evaluated UNIC on a workstation with an Intel Core i7-2600 CPU and 32GB RAM, running Fedora 20 with Linux 3.16.2. The cache server was running Redis 2.6.17. Our goal is to show that UNIC significantly improves performance with memoization while requiring minimal developers' effort and storage space.

The rest of this section focuses on three questions:

§3.5.1 Is UNIC easy to use?

§3.5.2 Does UNIC reduce computation time?

§3.5.3 What is UNIC's storage overhead?

3.5.1 Application Adaptation Effort

To evaluate whether UNIC is easy to use, we picked four popular open-source applications that we use daily: (1) clamav-0.98.1, an anti-virus software that scans a directory for viruses [30]; (2) pbzip2-1.1.8, a multi-threaded compression utility that compresses a single file [105]; (3) grep-2.18, a tool that searches for a regular expression within one or

Application	Total LoC	Changes	Percentage
clamav (file)	1,732,762	12	<0.01%
pbzip2 (block)	4,376	18	0.41%
grep (file)	9,658	35	0.36%
grep (block)	9,658	69	0.71%
gcc (file)	29,023	30	0.10%

Table 3.1: *Lines of code changed for each application.* Parenthesis indicates whether the adaptation uses file-level or block-level memoization. The numbers for gcc are based on ccache.

many files; and (4) the compiler gcc-4.8.3. We adapted them to use UNIC’s API². We used file-level memoization for grep, clamav, and gcc, and block-level memoization for grep and pbzip2.

Table 3.1 shows the lines of changed code for each application to use UNIC’s APIs. Changing dozens of lines (<1% of total lines) suffices for all these applications.

To further illustrate, we next present how we adapted grep, the application with the most code changes.

Case Study: grep

GNU grep is a line-based pattern searching utility. To invoke grep, the user specifies a search pattern and the path to a file or directory. Then grep iterates through all files in the directory and search for the pattern.

Common to all applications, the first step is to add a call to `init()` at the beginning of `main()` in order to initialize UNIC. For grep specifically, there are two design choices: we can memoize either at file-level or at block-level. Memoizing at file-level is faster when the whole file is unchanged, whereas memoizing at block-level can exploit sub-file

²Our adaptation of gcc is based on ccache [26].

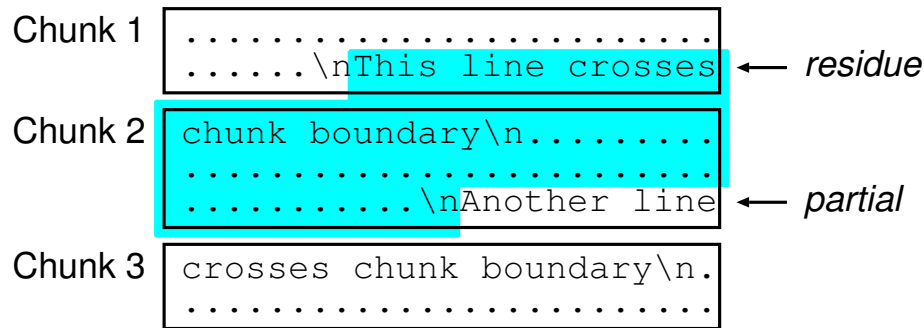


Figure 3.6: *Misalignment between line and chunk boundaries in grep.* Shaded region is the adjusted chunk for computation.

similarities for different files. Next we discuss each of them.

File-level Memoization. Adapting grep for file-level memoization is relatively straightforward. When grep works on a new file, we call `get_file_hash()` to get the hash of the file from ZFS and call `exists()` to check if there is a corresponding entry in the result cache. If so, we call `get()` to retrieve the memoized result, output it, and move on to the next file. If not, we follow the original algorithm and call `put()` to memoize whatever is output. We also call `put()` to memoize the number of matched lines in the current file, which grep uses for internal bookkeeping purposes.

Block-level Memoization. Adapting grep to memoize at block-level requires tighter integration with its workflow. For each file, grep reads its content in 32KB chunks, and performs pattern searching one chunk at a time. However, since the searching is line-based (delimited by ‘\n’), it is possible that lines are not well-aligned with chunk boundaries. For example, one line may span across the end of the previous chunk and continue at the following chunk. In this case, grep adjusts its chunk boundary to include the residue of the line in the previous chunk and exclude the partial line at the end of current chunk, as

shown in the shaded region in Figure 3.6.

Unfortunately, this poses a challenge to using UNIC directly, because ZFS keeps hash metadata only for entire aligned 32KB disk blocks. On the other hand, we cannot simply use the hash of the unadjusted chunk to address the cache, because this would err if two chunks were the same but their residues in the previous chunk differed. Our solution is to combine the hash of all chunks from the beginning of the residue until the current chunk. Note that this may lose the rare opportunity of reusing memoized results for chunks who only differ at the last partial line, but it preserves correctness nevertheless.

Our experience with adapting the other three applications were straightforward. Overall, we found UNIC easy to use and the adaptation effort was generally little.

3.5.2 Performance

To understand the performance of UNIC, we first use microbenchmarks to evaluate the throughput of UNIC's basic operations. We then run UNIC on four real-world applications to see how UNIC reduces application running time. Next, we study how UNIC is able to reuse previous computation results for some evolving data. Finally, we study how UNIC performs with a group of multiple users whose data are similar yet different.

Microbenchmark

We first use microbenchmarks to evaluate the throughput of the `get()` and `put()` operations. We wrote a program that calls `put()` 10,000 times followed by calling `get()` 10,000 times. The hashes of the 10,000 entries are all different, and we varied the result size from 1KB to 1MB.

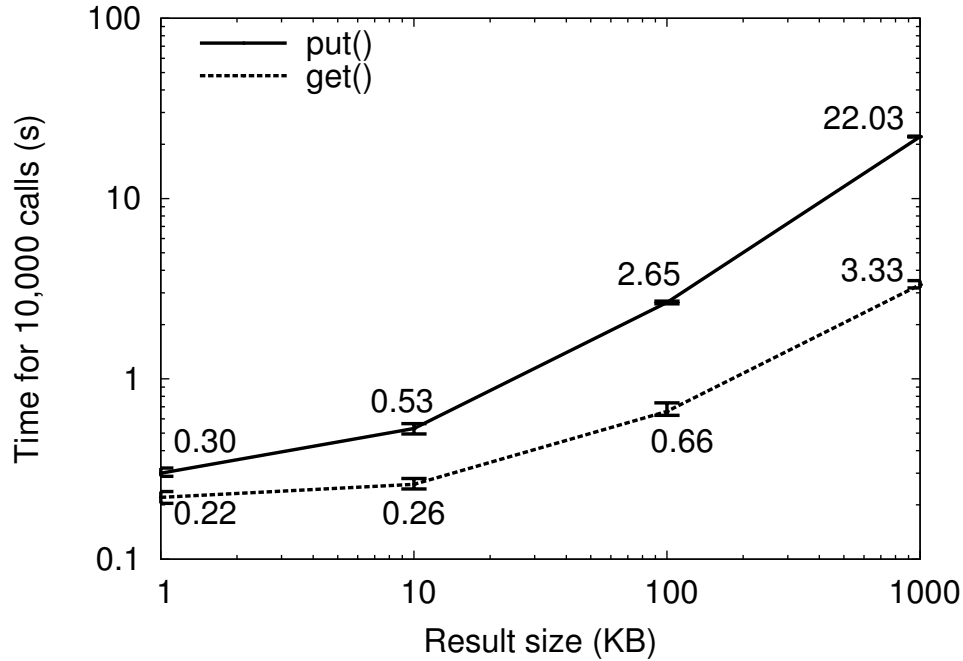


Figure 3.7: *Throughput of put() and get() operations.* The x -axis is the size of memoized result. The y -axis is the total time in performing 10,000 put() (solid line) and get() (dashed line) operations.

Figure 3.7 shows the results, where each data point is an average of 10 individual experiments with an error bar showing the maximum and minimum value in the 10 experiments. The x -axis is the size of the memoized result. The y -axis is the total time in performing the 10,000 operations. The solid line is for put() and the dashed line is for put(). From the results we find that the time for an operation is on the order of ten microseconds when the memoized result is small in size ($<10\text{KB}$), which is mostly the case (see §3.5.3). Even if the memoized result is as large as 1MB, the time to get a memoized entry is only 0.33ms, which is normally much faster than doing real computation on that size of data. Therefore, UNIC's basic operations are sufficiently fast for doing useful caching of computations.

Application Performance

We next show how real-world applications benefit from UNIC, and how storage deduplication further helps. We conducted the following experiments. (1) We used `clamav` to scan for viruses on two data sets. The first is the `linux-3.12` kernel source code tree. The second is the Dropbox folder for one of the co-authors, which contains 10.8GB of documents, music, pictures, videos, and applications. (2) We used `pbzip2` to compress `linux-3.12.tar` into `linux-3.12.tar.bz2`. (3) We ran `grep` on two data sets. The first is the `linux-3.12` kernel source code tree, which consists of 47,336 small files totaling 508MB. The second is the tags file of the `linux-3.12` kernel source code generated by `ctags -R`, which is a single text file of 250MB. For each data set, we ran a simple query ('void') and a complex query ('`^\s*struct\s+\w+\s+*\s*\w+\s*=\s*\w+\s*((\s+\w+(,)*\s)+\s);`;' for the source code tree, which matches declaring and initializing a structure pointer to the return value of a function, such as "`struct task_struct *task = get_proc_task(inode);`";, and '`/[A-Za-z]+\.\c.*d.*file`' for the tags file, which matches a specific type of tag). (4) We used `gcc` to compile `linux-3.12` kernel with the `allnoconfig` configuration. Because `gcc` has a nontrivial way to represent input dependencies for cache reusability rather than a file hash, our adaptation does not leverage storage deduplication information. All data files are on a freshly-formatted ZFS disk with cold buffer cache.

For each application, we compared the running time (1) without UNIC (the baseline), (2) with UNIC but without filesystem deduplication (the first and second bars on Figure 3.8), and (3) with both UNIC and filesystem deduplication support (the third and fourth bars). For experiments with UNIC, we further compared the running time (1) for execution

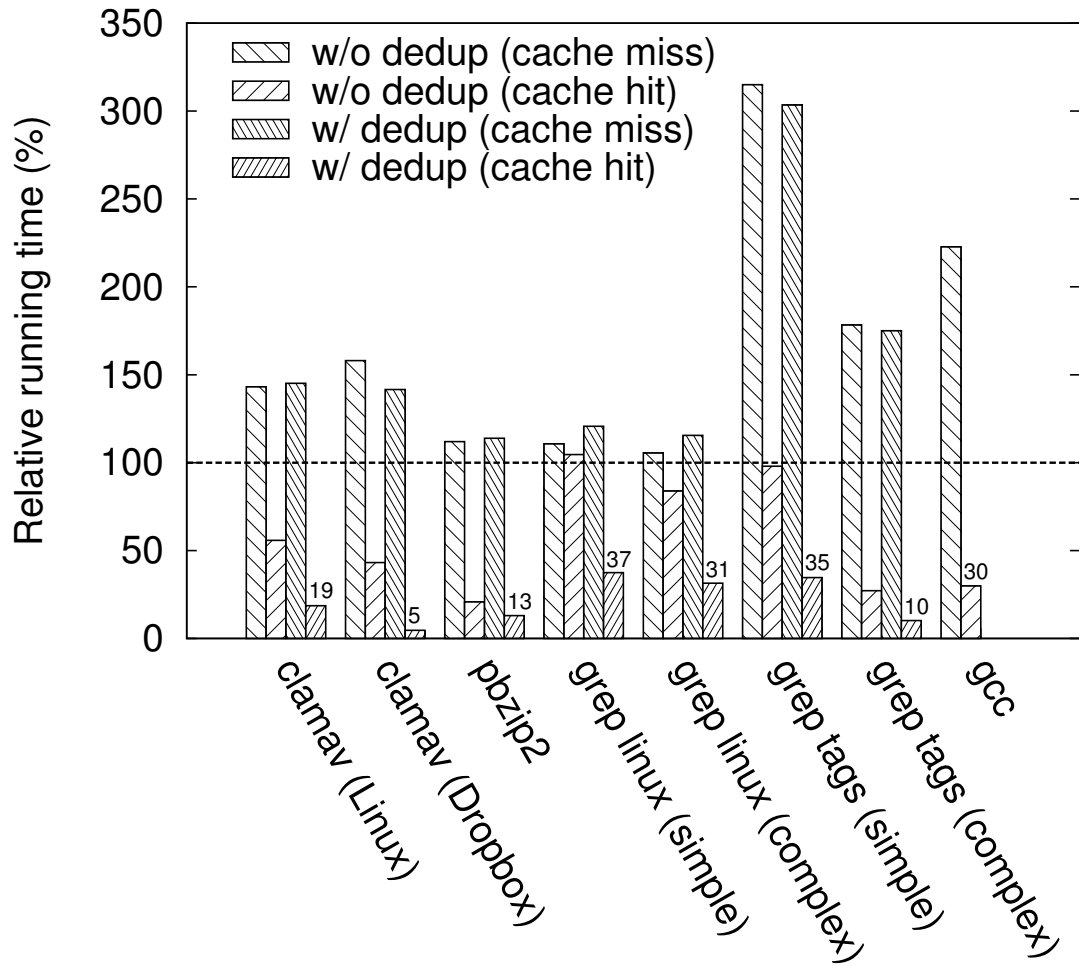


Figure 3.8: *Relative running time of applications.* The y -axis is the running time relative to the original application. For each cluster, the first bar is cache-miss execution without FS deduplication, the second bar is cache-hit execution without FS deduplication, the third bar is cache-miss execution with FS deduplication, and the fourth bar is cache-hit execution with FS deduplication. The dashed line at 100% shows the running time for the original application.

on an initially empty result cache, causing cache misses and thus putting entries to the cache (the first and third bars), and (2) for execution when the result cache had already been pre-populated, causing cache hits (the second and fourth bars).

Figure 3.8 shows the running time for each experiment. Each number is an average of 10 individual runs. Although running applications on an empty result cache incurs

an average overhead of 68.2%, running them on a warm result cache gives an average speedup of 2.39 \times . If filesystem deduplication is available, the average overhead of cache-miss execution drops to 59.3% and the average speedup with memoization increases to 7.58 \times . Furthermore, complex computations (*e.g.*, scanning for viruses or compressing a file) benefit the most from memoization (up to 21.4 \times speedup), while simple computations (*e.g.*, searching for a short string) suffer more from the cache-miss overhead. Therefore, opportunistically enabling memoization would be the best practice. With our strategy described in §3.4.2, memoization is enabled at the second occurrence of `put()` for one application (“grep tags” with simple query), and at the first occurrence for all other applications.

Effectiveness with Evolving Data

The previous evaluation focused on the memoization benefit on exactly the same computation. Next we show the effectiveness of memoization if the input data is evolving, *i.e.*, if UNIC has memoized computation on an old version of data, how it can speed up computation on a new version of the data.

We used `grep` to search for ‘void’ on thirteen major versions of the Linux kernel source code, from v3.0 to v3.12. All files are on a freshly-formatted deduplication-enabled ZFS disk with cold buffer cache. We performed three sets of experiments. The first one used the original `grep` without UNIC. In the second experiment, we first populated the result cache when running `grep` on v3.0, and then measured the time for running `grep` on each version based on the same memoization of v3.0. In the third experiment, we ran `grep` on each version in a “rolling” manner, *i.e.*, each execution was based on the memoization

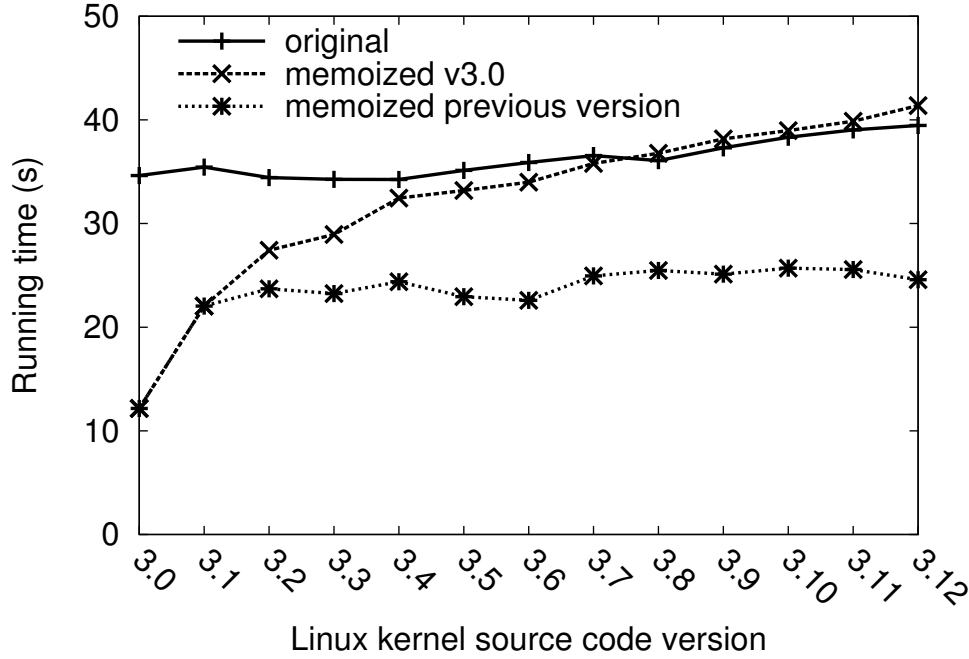


Figure 3.9: *Effectiveness of memoization with evolving data.* Solid line is the original grep without memoization. Dashed line has the result cache populated with v3.0. Dotted line has the result cache populated with the immediate previous version.

of the immediate previous version, which resembles a more practical scenario.

Figure 3.9 shows the running time for all executions, where each number is an average of 10 runs. With a single memoization of v3.0, the speedup is significant for running on v3.1 (1.61 \times), but diminishes along the increment of version number, and eventually becomes ineffective after v3.8, because the source code differs significantly from the memoized version and the cache hit rate drops below 0.3. On the other hand, when memoized the immediate previous version, the speedup is almost constant, with an average of 1.50 \times . The reason is that the amount of source code difference is almost constant between each two consecutive versions, and many memoized results can be reused (hit rates are between 0.73 and 0.81). Therefore, UNIC is more effective when the divergence of the actual input data from the memoized data is small, which is likely true in a practical scenario.

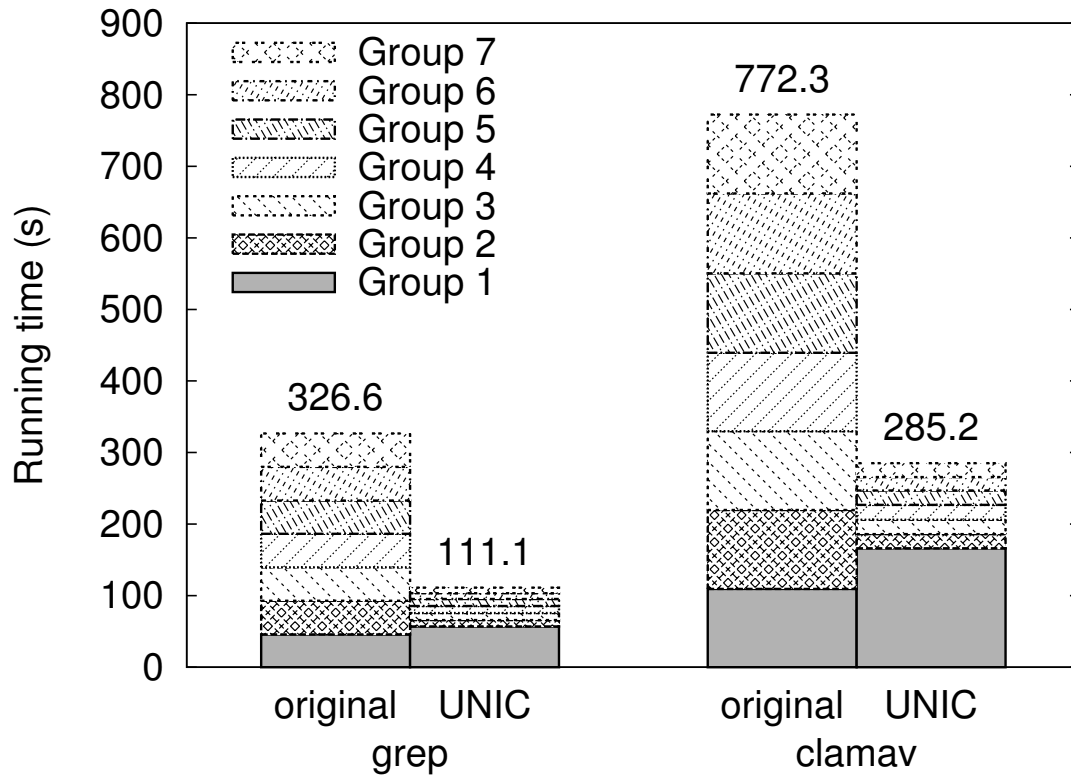


Figure 3.10: *Effectiveness of memoization across users.* For each cluster, the first bar is the original application, and the second bar is the application modified to use UNIC. Each bar shows the breakdown of running time on each group, the number on top showing the total time.

Effectiveness with Multiple Users

We next evaluate the memoization effectiveness for multiple users with similar yet different data. We took the project directories of seven groups of students in a graduate-level operating system course offered by our university. The average size of each directory is 1.6GB. We performed two executions on each group’s directory: (1) use `grep` to search for ‘void’, and (2) use `clamav` to scan for viruses. This resembles the enterprise setting where multiple people working on the same project have similar data and perform common computing tasks such as virus scanning. The result cache was originally empty, and was gradually filled by UNIC during the process.

Figure 3.10 shows the breakdown of each application’s running time on each group. The trend is that the original application takes almost the same amount of time for all groups. With UNIC, although the first group takes longer time to execute (24.1% for `grep` and 51.9% for `clamav`), all subsequent groups consistently take a much shorter time ($5.17\times$ speedup for `grep` and $5.57\times$ speedup for `clamav`). This is because for the first group, all computations are new and UNIC needs to insert them to the result cache. Once this is done, all subsequent groups can benefit from it. The overall speedups for the executions on all seven groups are $2.94\times$ for `grep` and $2.71\times$ for `clamav`. We foresee that with more number of groups the overall speedup should be even higher. Therefore, UNIC is practical for a group of users working together or doing similar tasks.

3.5.3 Storage Space

We now evaluate the storage overhead of UNIC. For each application we used for the performance evaluation in §3.5.2, we examined the number of entries in the result cache. To study the total space used for memoization, we also let Redis dump a snapshot of all data and measured the size of the dump file.

Table 3.2 shows the results. Column (a) is the number of input files. Column (b) is the total size of input files. Column (c) is the number of entries in the result cache. Column (d) is the size of the Redis dump file. The relative storage overhead is thereby Column (d) divided by Column (b), which is shown in Column (e). The results depict that the average overhead of the memoization storage for all applications is 3.45%, negligible compared with the storage of all file data. Therefore, UNIC incurs little storage overhead.

Application	(a) #File	(b) File size	(c) #Entry	(d) Dump size	(e) Overhead
clamav (Linux)	47,336	508.1MB	44,277	2.8MB	0.55%
clamav (Dropbox)	2,792	10.8GB	82,061	4.4MB	0.04%
pbzip2	1	544.0MB	4,151	106.4MB	19.55%
grep linux (simple)	47,336	508.1MB	70631	11.2MB	2.21%
grep linux (complex)	47,336	508.1MB	51532	4.2MB	0.83%
grep tags (simple)	1	250.0MB	2	5.3MB	2.13%
grep tags (complex)	1	250.0MB	2	4.5MB	1.80%
gcc ³	47,336	508.1MB	522	2.3MB	0.46%

Table 3.2: *Storage overhead*. Columns are: (a) the number of input files, (b) total size of input files, (c) number of entries in the result cache, (d) size of the Redis dump file, and (e) relative storage overhead.

3.6 Discussion and Limitations

We discuss UNIC’s security implications and limitations.

Denial-of-service attacks. A malicious user may issue a large number of put requests on manufactured inputs, and pollute the result cache with useless results. Several approaches can be used to defend against it. For example, UNIC may rate-limit puts to the result cache, employ a quota mechanism to limit the cache space for each client application, or enforce time-to-live limits on cached results. We argue that even if the result cache is full, the worst outcome would be that future computations cannot be memoized and have to be recomputed, yet the secrecy and integrity of computations are not violated.

Side-channel information leakage. A malicious user may enumerate through a large set of inputs on an application, and observe if some executions are significantly faster than others. Based on the observed timings, she may infer what computations have been done by other users and what have not. While defending against this side-channel attack is out of the scope of this dissertation, we note that the application developers may defend

against it by rate-limiting queries to the result cache or randomly forcing cache misses even if the result exists in the cache.

Brute-force attacks. A malicious user may enumerate through all possible hash values of the application code and input, in hopes of getting cached results. We argue that the possibility for an unprivileged user to get a valid hash is minimal. Even if she manages to get an entry, she only knows the result, but she cannot generate the original code and input from the hash. In the example of virus scanning, she might brute-force a hash and discover the result of scanning some file, but she cannot determine the original content of that file. Again, UNIC may defend against this attack by rate-limiting queries to the result cache. Furthermore, if the result is sensitive by itself (*e.g.*, cat), the application developer may encrypt it before putting it to the result cache, or the system administrator may disable UNIC for such applications.

Application bugs. Ensuring bug-free code is a hard problem orthogonal to UNIC and code attestation. If the application contains a bug such as buffer overflow, a malicious user may exploit the bug to poison the result cache. Existing systems such as baggy bounds checking [5] and AddressSanitizer [121] can prevent many memory access bugs. Other countermeasures include letting the application rerun the computation and verify the cached result periodically, and purging the result cache when a bug is found. In addition, using hardware-enforced isolation mechanisms such as Intel TXT [76] with TPM, or Intel SGX [78, 16] may avoid this issue.

³Not all files are used for compilation due to our experiment configuration.

3.7 Related Work

Storage deduplication. Storage deduplication reduces data redundancy at either file-level [92] or block-level [42, 133]. ZFS [150] is a widely used cross-platform filesystem that does block deduplication at the time data is written. These works are orthogonal to UNIC, and UNIC’s cross-layer design allows it to transparently leverage storage deduplication information.

Ad-hoc caching. Many applications use ad-hoc caching to improve performance, but they either trust all users, or simply disallow cross-user caching. For example, ccache [26] caches compiler outputs on the local filesystem, but the cache can be easily exploited or poisoned by any user. On the other hand, clamav [30] only caches virus scanning results within a single session, rendering cross-session and cross-user caching impossible. UNIC improves the status quo with strong security guarantees.

Memoization. Memoization [93, 114, 87] is a technique that reuses prior computation results of functions without side effects. Vesta [71] uses memoization for software configuration management. Nectar [69] memoizes intermediate results from DryadLINQ [149] programs. Incoop [21] uses memoization to build a MapReduce framework for incremental computations. However, these systems handle only specific computations, and it is nontrivial to generalize their use cases. UNIC can be used to deduplicate general computations.

Code attestation. Many code attestation techniques exist to provide integrity of computations. For example, result-checking [139] verifies the result produced by a program

by computing it in two ways. Secure boot mechanisms [13, 14] verify the integrity of the software stack after booting. BIND [125] ties the proof of what computation has been run to the result that the computation has produced. Pioneer [124] provides code integrity guarantees for running software on an untrusted system. UNIC makes novel use of the code attestation mechanism to protect the secrecy and integrity of memoization.

3.8 Summary

This chapter presented UNIC, a general system for applications to securely deduplicate their rich computations. It uses code attestation mechanism to achieve both secrecy and integrity. It explores a cross-layer design that allows applications to leverage storage deduplication information for speed. Evaluation results show that UNIC is easy to use, speeds up applications by up to 21.4×, and incurs little storage overhead.

Chapter 4

Optimizing Serverless Computing by Making Data Intents

Explicit

Serverless computing emerges as a new paradigm to build cloud applications. Developers write small functions, called *cloud functions*, that react to cloud infrastructure events, while the cloud provider maintains all resources and schedules the functions in containers. Thus, developers can focus on their core business logic, and leave server management and scaling to the cloud providers.

Besides ease of programming, serverless computing provides more efficient and fine-grained scaling than traditional clouds, because containers are more lightweight than virtual machines. It adjusts to dynamic workload at a per-function level and scales up or down in a second.

As a result, many companies, including Netflix, Coca-Cola, and the New York Times, are adopting serverless computing [35]. A 2018 survey of 600 IT decision-makers shows that 61% of respondents are already using or plan to use serverless computing by 2020 [57]. Practically, all major cloud providers provide serverless computing services, namely AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions.

Unfortunately, existing serverless computing systems suffer from a key limitation that deprives them of enjoying significant speedups. Specifically, they treat each cloud func-

```
thumbnail(params={  
    "get_data": ["pic/1.jpg"],  
    "put_data": ["thumb/1.jpg"]  
})
```

Figure 4.1: *Example of specifying data intent for a cloud function that generates the thumbnail of an image. It reads input from `pic/1.jpg` and writes output to `thumb/1.jpg`.*

tion as a black box and are blind to which data the function reads or writes, therefore missing potentially huge optimization opportunities. For instance, they schedule multiple functions working on the same data to run on different machines, neglecting data locality. Thus, each machine has to fetch a copy of the data, resulting in a 42% slowdown and 19.2% more monetary cost.

We present LAMBDATA, a novel serverless computing system that enables developers to declare a cloud function’s data intents, including both data read and data written. Figure 4.1 shows an example that a thumbnail function intents to read `pic/1.jpg` and write `thumb/1.jpg`. Once data intents are made explicit, LAMBDATA performs a variety of optimizations to improve speed, such as colocating functions working on the same data. These intents are hints only: if a developer misses an intent or specifies an incorrect one, performance may be affected but not correctness.

Our design of LAMBDATA is strongly motivated by two key insights in serverless computing. First, a cloud function is almost always small doing a single task; therefore, developers can easily specify its inputs and outputs before executing it, as illustrated in Figure 4.1. In other words, the paths to input and output data are typically not calculated on the fly amid the execution of a cloud function. While current serverless clouds allow a developer to write a large monolithic function that dynamically computes data locations

and accesses the data, such use would defeat the main benefit of serverless computing.

Second, based on our study of open-source serverless applications and our own experience building such applications, a cloud function tends to be functional in the sense that it outputs an immutable object: once the object is written, the application does not mutate it. If an update is needed, the application simply writes a new object under a new path. This approach avoids complicating cloud functions with tricky concurrent and partial update handling logic, and it is a natural fallout from the idempotency requirement of the underlying serverless cloud (the cloud may kill and restart a cloud function without notification due to resource constraints or tail latencies). This insight enables LAMBDATA to aggressively cache data objects throughout the system to improve locality without concerning consistency issues.

Operationally, LAMBDATA works as follows. It leverages existing cloud object storage (e.g., AWS S3) to store data. LAMBDATA adds a caching layer, where each computing node has its own object cache. LAMBDATA schedules cloud functions based on both code and data locality. It tends to schedule multiple function invocations working on the same data on the same computing node so that they can reuse cached data.

Compared with Pocket [82], we choose to build LAMBDATA on top of existing cloud object storage. The benefits are two folds. First, using cloud storage is the best practice recommended by Amazon [122] and Google [63], since data objects enjoy the high durability they offer. Second, developers are familiar with this programming model, since they do not need to decide what data should be durable and what data can be put on ephemeral storage.

Our evaluation of LAMBDATA on an Instagram-like application and an online class-

room application shows that on average, LAMBDATA achieves $1.51\times$ speedup on the turnaround time of practical workloads and reduces monetary cost by 16.5%.

The remainder of this chapter is organized as follows. The next section introduces the background of serverless computing. §4.2 motivates LAMBDATA’s design through an example. §4.3 gives an overview of LAMBDATA. §4.4 describes LAMBDATA’s data-aware scheduling algorithm. §4.5 describes the optimization of direct file access. §4.6 shows evaluation results. §4.7 discusses some design implications. §4.8 presents related work, and §4.9 summarizes this chapter.

4.1 Background: Serverless Computing

In serverless computing, the basic building block is a function. A cloud function is similar to a function in a computer program, in that it takes some parameters, performs a task, and returns a result. A cloud function can be triggered by another function, by a RESTful API, or by a cloud event, such as when the cloud storage receives a new file or a database gets a new entry.

With serverless computing, developers do not need to manage any infrastructure. The cloud service provider handles all resource management. It runs a function in a container, and each container is isolated from one another. When a function ends, its container is paused for a few minutes before being terminated. If the same function gets invoked again while the container is paused, the same container will be resumed, which we call a warm start. Otherwise, it is a cold start.

Cloud functions cannot rely on containers to persist any state, because containers are

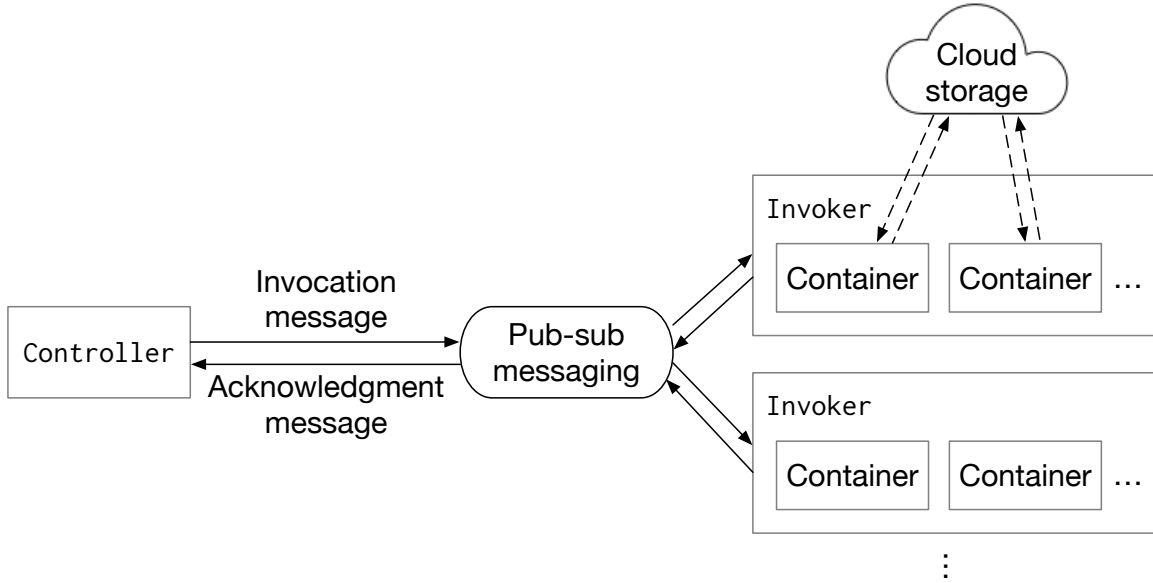


Figure 4.2: *Overview of serverless architecture.*

ephemeral. Cloud service providers also limit the size of function parameters and return values to a few hundred kilobytes, making it impossible to pass large data this way. As a result, cloud functions have to leverage cloud storage services (e.g., AWS S3) to store or pass any non-trivial data.

The price for using serverless computing services typically consists of two parts: a flat-rate cost per function invocation (“request cost”), plus a cost proportional to the function’s run time (“duration cost”). The request cost is only a fraction of the duration cost. Hence, it is desirable to minimize the function run time.

4.1.1 Overview of serverless architecture

A typical serverless cloud (e.g. Apache OPENWHISK [103, 31]) consists of two fundamental entities: one Controller and multiple Invokers (Figure 4.2). The Controller is the orchestrator of the system, and the Invokers are the executors. They communicate through

a publish-subscribe messaging system (e.g., Apache KAFKA [81]).

To invoke a function, the Controller schedules the invocation to run on an Invoker and publishes an *invocation message*. When the Invoker receives an invocation message, it publishes an *acknowledgment message*, and starts or resumes a container to run the function.

4.1.2 Life of a cloud function

A typical lifecycle of a cloud function consists of four phases: start, get, compute, and put.

The start phase is starting up the function. For a cold start, the cloud starts a new container, downloads the function code to the container, and invokes the function. The function may then install additional packages (e.g., OpenCV, FFmpeg, or L^AT_EX) or make one-time network connections (e.g., to a database). For a warm start, the cloud resumes an existing container and invokes the function. A warm start takes less than 20ms, while a cold start usually takes more than 1s, depending on the function.

The get phase is getting the input data from the cloud storage service. The typical time spent on getting the data is between 100ms and 5s, depending on the data size.

The compute phase is performing the actual computation on the data. Although different functions have vastly different computations, most functions are quick tasks that finish within 3 seconds, the default time limit on AWS Lambda.

The put phase is putting the output data back to the cloud storage. Different functions generate different sizes of data, which usually takes between 100ms and 5s to upload. A special case is if a function performs a classification task, such as malware detection

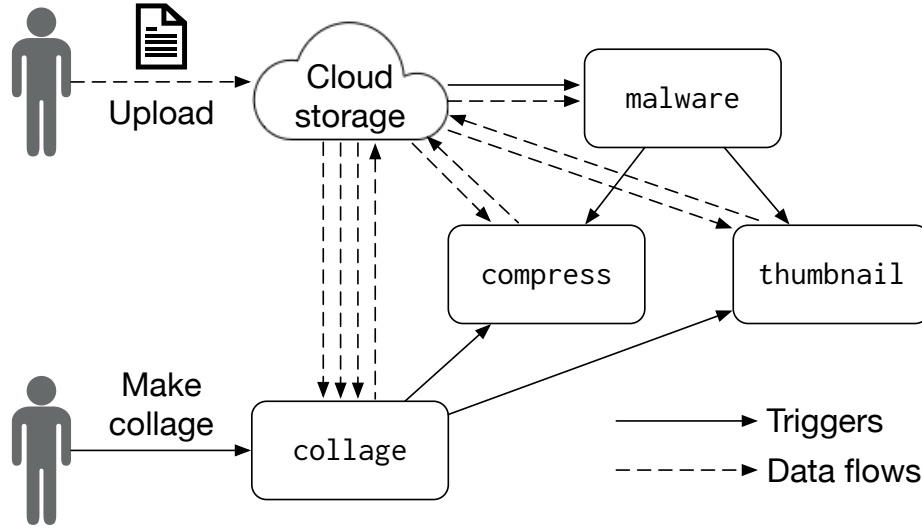


Figure 4.3: A photo-sharing application example. Solid arrows indicate triggers. Dashed arrows represent data flows.

or image recognition, it usually leverages the function’s return value or uses a database service (e.g., DYNAMODB) instead, without putting data back to the cloud storage.

4.2 A Motivating Example

4.2.1 Example and insights

We motivate the design of LAMBDATA through an example of a photo-sharing application with four cloud functions: `malware`, `compress`, `thumbnail`, and `collage`. We compose these functions into two major workflows: *handling user upload* and *making collage*. Figure 4.3 shows the triggering of functions and how the data flows in and out of the cloud storage. Figure 4.4 shows the source code that handles each function.

Handling user upload. The user uploads an image using a front-end application (e.g., a smartphone app), which puts the image on the cloud storage. As the cloud storage

```

def malware(params):
    bucket, key = params["get_data"][0]
    file = get(bucket, key)
    result = do_scan_malware(file)
    return {"result": result}

def compress(params):
    bucket, key = params["get_data"][0]
    image = get(bucket, key)
    small_image = do_compress_image(image)
    put_bucket, put_key = params["put_data"][0]
    put(put_bucket, put_key, small_image)

def thumbnail(params):
    bucket, key = params["get_data"][0]
    image = get(bucket, key)
    thumbnail = do_generate_thumbnail(image)
    put_bucket, put_key = params["put_data"][0]
    put(put_bucket, put_key, thumbnail)

def collage(params):
    images = []
    for bucket, key in params["get_data"]:
        images.append(get(bucket, key))
    collage = do_generate_collage(images)
    put_bucket, put_key = params["put_data"][0]
    put(put_bucket, put_key, collage)

```

Figure 4.4: *Example code with four cloud functions.*

receives the data, it automatically triggers malware for next-stage processing. Figure 4.5 shows the data flow graph.

Function `malware` is a malware-detection module. It is triggered by a file-upload event of the cloud storage service. When triggered, this function fetches the data from the cloud storage and runs a malware-detection program. If the file is clean, then it triggers both `compress` and `thumbnail` simultaneously for next-stage processing. Otherwise, it discards the file.

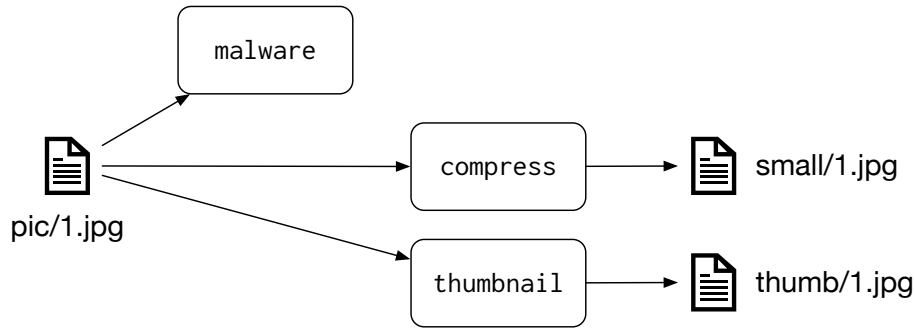


Figure 4.5: *Data flow of handling user upload.*

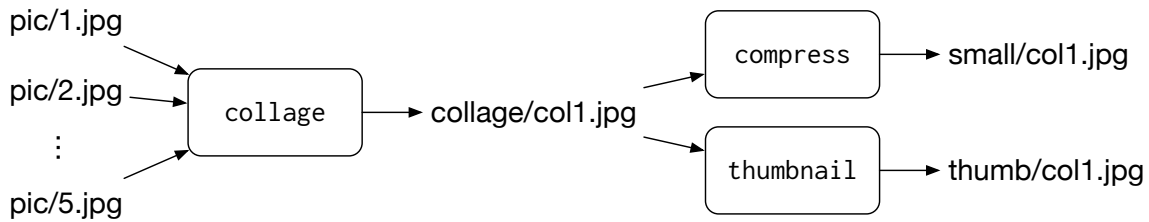


Figure 4.6: *Data flow of making collage.*

Function `compress` gets an image file from the cloud storage, compresses it, and puts it back to the cloud storage. Similarly, Function `thumbnail` gets an image from the cloud storage, generates a thumbnail, and puts it back to the cloud storage.

Making collage. The user can also make a collage out of several existing images. Figure 4.6 shows the data flow graph. The front-end application sends a REST request to the cloud gateway, which triggers `collage` with a list of image keys. Function `collage` gets each image file from the cloud storage, generates a collage image, and puts it back to the cloud storage. It also triggers `compress` and `thumbnail` to compress the collage and generate a thumbnail of the collage.

Insights. From the example, we observe two key insights in serverless computing. We have also studied over a dozen open-source serverless applications on Github and built

	Input	Output
<i>Workflow: handle user upload</i>		
malware	pic/1.jpg	none (return value only)
compress	pic/1.jpg	small/1.jpg
thumbnail	pic/1.jpg	thumb/1.jpg
<i>Workflow: make collage</i>		
collage	pic/1.jpg—pic/5.jpg	collage/col1.jpg
compress	collage/col1.jpg	small/col1.jpg
thumbnail	collage/col1.jpg	thumb/col1.jpg

Table 4.1: *Inputs and outputs of each function.*

two serverless applications ourselves.¹ All these applications share the same insights.

Our first insight is that a cloud function is almost always small doing a single task; therefore, developers can easily determine its inputs and outputs before executing it, rather than calculate the object names on the fly. For example, Table 4.1 shows an example of inputs and outputs of each function.

If a function is triggered by a cloud storage event, then the input is just the object that emits the event. For example, in the Upload workflow, malware is triggered by the cloud storage when it receives a new image (e.g., `pic/1.jpg`), so the input is just `pic/1.jpg`. The developer can easily calculate the output objects deterministically before executing the function. Function malware only appends an entry in the database and does not generate new data objects, so the output is empty. If the function wrote to the cloud storage instead of the database, then it would specify something like `result/1.txt` as the output.

If a function is triggered by another function or a REST request, then the developer can specify the inputs and outputs based on her intent of invoking the function. For example,

¹Because serverless computing is a relatively new paradigm, we could only find a few real-world open-source serverless projects beyond demos and tutorials.

in the Collage-making workflow, the front-end application invokes `collage` via a REST request to combine a list of images into a collage. So the inputs are the list of image objects (`pic/1.jpg...pic/5.jpg`), and the output is the intended filename of the collage object (`collage/collage1.jpg`). Function `collage` further invokes `compress` and `thumbnail` to compress the image and generate a thumbnail, so it specifies the collage file as the input, and the outputs are just the same filename prepended with buckets `small/` and `thumb/`, respectively.

Our second insight is that a cloud function tends to be functional in the sense that it outputs an immutable object: once the object is written, the application does not mutate it. Because the serverless computing providers may kill a function or run a function more than once without any notice, they require that all functions be idempotent. Therefore, most developers write functions in a purely functional way, so that it is much easier to reason about the behaviors and handle failures.

For example, all of our four functions are purely functional, in that they never mutate data, and always generate the same output for the same input. Specifically, the image-compression function does not modify the input object in-place but rather writes the result as a new object. Otherwise, if the function is invoked twice, it would end up with double-compressing the image.

These two insights enable LAMBDATA to cache data aggressively without worrying about data inconsistency and schedule function invocations by considering both code and data locality.

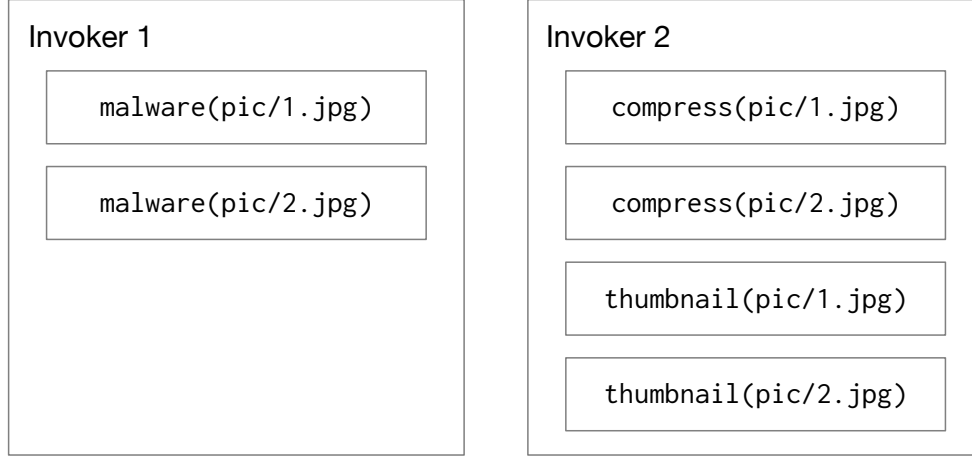


Figure 4.7: *Inefficient scheduling on OPENWHISK.*

4.2.2 Inefficiencies with existing serverless clouds

Existing serverless clouds regard a function as a black box, and treat all invocations of a function in the same way. When scheduling functions, they only consider the function code, but not the data that the function computes. As a result, they tend to schedule multiple invocations of the same function on the same Invoker. For example, we ran the workload of handling user uploads on two images, using OPENWHISK with two Invokers. Figure 4.7 shows the scheduling of all function invocations: the first Invoker handles all invocations of `malware`, and the second Invoker handles all invocations of `compress` and `thumbnail`, because this schedule is optimized for reusing warm containers.

Unfortunately, this scheduling is inefficient, because functions in both Invokers need to get both images from the cloud storage. As cloud functions are typically small, the time a function spent on getting data is significant. Table 4.2 shows that the functions in our example spend 40% of the time getting duplicate data from the cloud storage. Besides wasting time, it also costs more money because the cloud storage charges for each request.

Function	start	get	compute	put	%(get)
malware	402	208	69	n/a	44%
compress	132	216	117	83	39%
thumbnail	137	201	79	48	40%

Table 4.2: *Time spent on each phase of the functions, in milliseconds.* The last column shows the percentage of time spent on getting the data from the cloud storage.

The fundamental cause of this inefficiency is that existing serverless clouds have no way of knowing a function’s data intents (*i.e.*, what data the function needs to read and write); therefore, they cannot leverage such information for scheduling.

4.2.3 LAMBDATA’s optimizations

LAMBDATA optimizes for this inefficiency by making a cloud function’s data intents explicit. Developers or cloud events can easily annotate the input and output data when invoking a function, using two special fields in the function’s parameter list: `get_data` for input and `put_data` for output. These annotations enables the serverless cloud’s Controller to see through the black box when scheduling a function invocation. For example, the previous two invocations of `thumbnail` now look different with LAMBDATA:

```
thumbnail(params={
    "get_data": ["pic/1.jpg"],
    "put_data": ["thumb/1.jpg"]
})
thumbnail(params={
    "get_data": ["pic/2.jpg"],
    "put_data": ["thumb/2.jpg"]
})
```

})

Under the hood, LAMBDATA securely caches data locally on each Invoker, and the Controller takes into account both code and data locality when scheduling function invocations. We noticed three common data usage patterns of cloud functions while studying open-source serverless applications and writing our own applications, and designed LAMBDATA accordingly.

Temporal locality of data. A data is often reused by multiple functions within a short period of time. For example, in the workflow of handling user upload, immediately after malware finishes computation on a file, both compress and thumbnail perform computations on the same file concurrently. LAMBDATA securely caches the file locally on the Invoker, and schedules all three function invocations on the same Invoker according to their data intents (if it deems worthy, see §4.4). Therefore, only malware needs to get the data from the cloud, while compress and thumbnail read the cached data, reducing both time and monetary cost.

Spatial locality of data. Many tasks or workflows involve multiple functions computing on a small set of closely-related data. For example, a user often uploads several images in a row and then creates a collage from these images. Therefore, even a small cache provides many benefits.

Data pipelining. Multiple functions often process data as a pipeline. For example, in the collage-making workflow, functions collage and thumbnail form a pipeline, *i.e.*, collage triggers thumbnail, and the output of collage directly becomes the input of

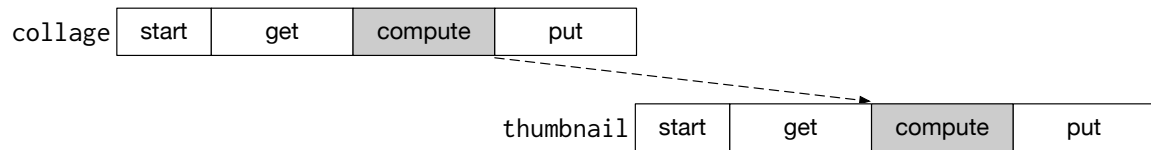


Figure 4.8: *Dependency between collage and thumbnail.*

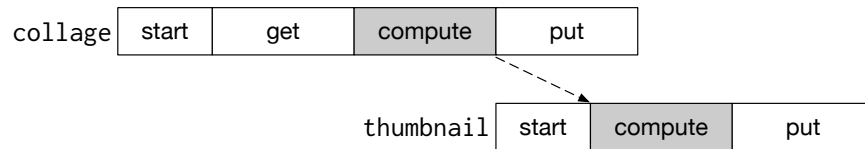


Figure 4.9: *Overlapping functions in a pipeline.*

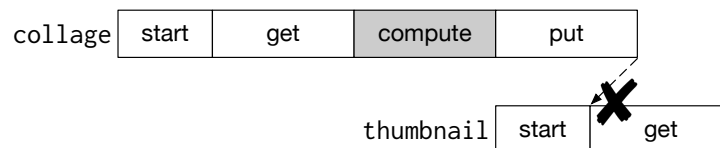


Figure 4.10: *A rare case of overlapping functions in a pipeline.*

thumbnail. With existing serverless clouds, thumbnail cannot start until collage finishes putting the data to the cloud storage. However, the real dependency between the two functions only lies in the actual computation of the data (the *compute* phase), as shown in Figure 4.8. Because LAMBDATA caches both input and output data, it schedules the next-stage function on the same Invoker as soon as the previous function enters the *put* phase (if worthy, see §4.4), effectively overlapping functions in a pipeline. Figure 4.9 shows LAMBDATA’s scheduling for this example: thumbnail starts when collage finishes computing, and it gets the collage output from the cache rather than going through the cloud storage.

In the rare case, if LAMBDATA schedules thumbnail on a different Invoker but thumbnail enters the *get* phase before collage finishes the *put* (Figure 4.10), then LAMB-
DATA would fail this invocation and retry it. Because serverless computing providers may

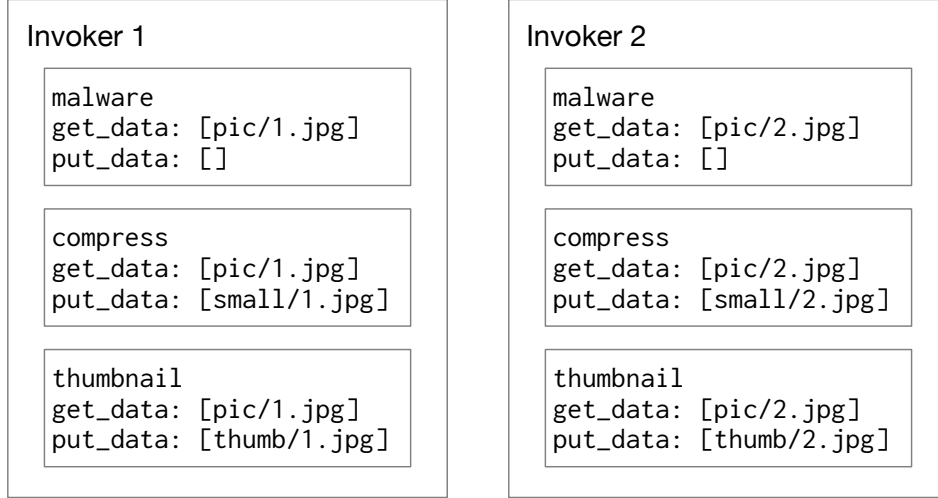


Figure 4.11: *Optimized scheduling with LAMBDATA.*

kill and restart a function without notice, and thus require all cloud functions to be idempotent, LAMBDATA’s behavior does not impose any new limitations.

We ran the same workload from §4.2.2 on LAMBDATA, and got the scheduling shown in Figure 4.11: the first Invoker handles all invocations related to 1.jpg, and the second Invoker handles those related to 2.jpg. As a result, LAMBDATA reduced the overall turnaround time by 29.6%, a 1.42 \times speedup, and reduced the monetary cost by 19.2%. We show more case studies in §4.6.

4.3 LAMBDATA Overview

We give an overview of LAMBDATA’s API and architecture.

4.3.1 LAMBDATA API

LAMBDATA exports a simple yet effective API for serverless functions to deal with cloud storage systems.

Basic cloud storage methods. LAMBDATA's API for basic cloud storage methods resembles the cloud storage's original API, such as `get` and `put`.

```
object = get(bucket, key)

put(object, bucket, key)
```

Developers simply import the LAMBDATA library and use these methods in the same familiar way.

Data intents. LAMBDATA lets a function specify in its parameters what data it needs to get and put. A function in current serverless computing services represents all parameters as a single JSON string. LAMBDATA inserts three new fields `get_data`, `put_data`, and `num_threads` into the JSON. The developer specifies by `get_data` and `put_data` all data it needs to get and put, both as an array of (bucket, key) pairs, and by `num_threads` the number of threads it uses to connect to the cloud storage for parallel downloads. Figure 4.12 shows an example annotation of invoking the collage function to make a collage from two images, using up to 5 threads for getting the data. Therefore, the scheduler knows each function's data intents by peeking at the JSON string, before running the function.

We note that many existing serverless functions already include an equivalent of `get_data` and `put_data` in their parameters, but there is no standard on how they would name these fields. For those functions, developers simply need to change the names of these fields into `get_data` and `put_data`, and enjoy the benefits of LAMBDATA.

All annotations are hints only. If a developer misses an intent or specifies an incorrect one, performance may be affected but not correctness. For example, if `get_data` is missing, LAMBDATA may not schedule the function on the best Invoker, but the func-

```

{
  "get_data": [
    {
      "bucket": "pic",
      "key": "1.jpg"
    },
    {
      "bucket": "pic",
      "key": "2.jpg"
    }
  ],
  "put_data": [
    {
      "bucket": "collage",
      "key": "col1.jpg"
    }
  ],
  "num_threads": 5
}

```

Figure 4.12: *Example annotations for an invocation of collage.*

tion can still opportunistically benefit from cached data. If `put_data` is missing, LAMB-
DATA can provide all benefits except that it would not overlap functions in a pipeline. If
`num_threads` is missing, LAMBDATA assumes the function gets data sequentially.

4.3.2 Architecture

We modified OPENWHISK and added LAMBDATA functionalities to several components.

Figure 4.13 shows the architecture.

Controller. We implement a data-aware scheduler in the Controller, which consists of
a cache registry, a data size registry, and some profiling results. The cache registry keeps a
list of cached data keys for each Invoker, possibly stale. The data size registry maintains
the size of all data objects. The profiling results contain the recent performance of the

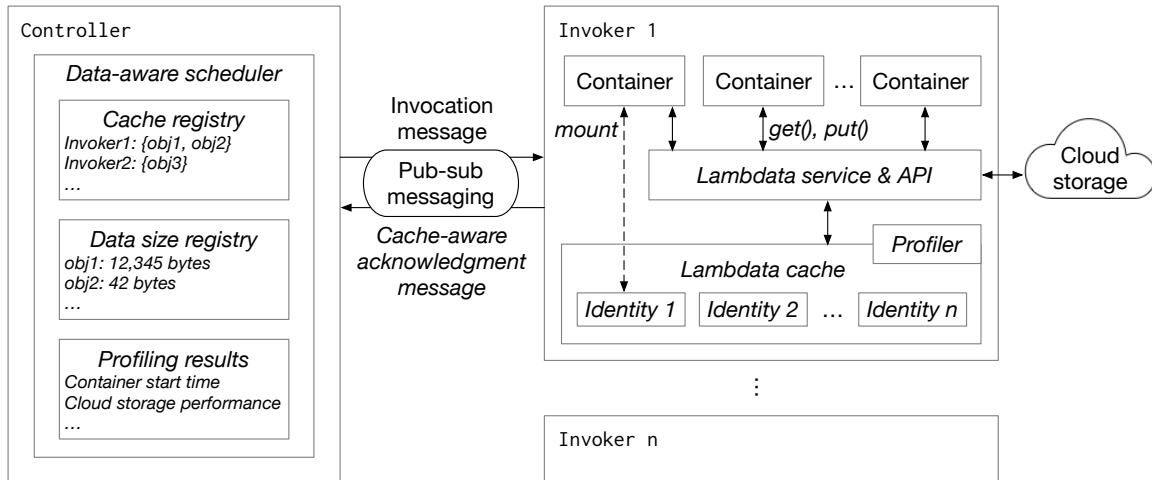


Figure 4.13: *LAMBDATA's architecture*. Components with italic font are LAMBDATA-specific.

cloud, including the time to start a container, the time to get and put a file from the cloud storage. The Controller does not actively probe any information, but only bookkeeps information sent by Invokers.

Invoker. Each Invoker independently manages its own cache and sends a list of all currently cached data keys and the size of each data object to the Controller by piggybacking it with the *acknowledgment message* of each function invocation. It also monitors the time to resume a warm container or start a cold container for each function. Whenever it gets or puts a data object to the cloud storage, it monitors the time it takes, in order to estimate the time of cloud storage operation for various data sizes. It sends these profiling results to the Controller via the acknowledgment message, too.

Each Invoker exports the LAMBDATA API via Unix domain socket. Each container on the Invoker has access to the API socket, and optionally mounts a portion of the cache that the function's identity has access to (§4.5).

4.4 Data-Aware Scheduling

In existing serverless cloud architectures such as OPENWHISK, each Invoker individually manages containers, and the Controller does not know where the warm containers are. Thus, the Controller only uses a deterministic hash of the function to pick an Invoker. Although this method has a good chance of picking a warm container, it fails to consider data locality.

By contrast, LAMBDATA employs a data-aware scheduling algorithm. Among the four phases of a serverless function, the compute and put phases are essential computations, unaffected by scheduling. Therefore, LAMBDATA’s scheduling algorithm aims to pick an Invoker in order to minimize the *lead time*, the time spent in the start and the get phases. We denote them by T_{start} and T_{get} , respectively.

Unfortunately, T_{start} and T_{get} do not always align. For example, consider two Invoker candidates, one with a warm container but no cached data, the other with cached data but no warm containers. Scheduling the function on the first Invoker means a small T_{start} but a large T_{get} , while scheduling on the second Invoker means a large T_{start} but a small T_{get} . In order to minimize $T = T_{\text{start}} + T_{\text{get}}$, we need to estimate both times.

4.4.1 Estimating the lead time

The start phase. Let us name the function f . We denote by T_{warm} the time to resume a warm container, and notice that it is fast regardless of what function is in it. We denote by T_{cold} the time of Phase 1 if we have to start a new container, and find that it is relatively stable for the same function but varies greatly from function to function, so we model

it as a function of f . We notice that T_{cold} is not determined by the static size of the f that the developer submits because some functions install additional packages or make one-time network connections after the initialization of the container. To deal with this issue, LAMBDATA monitors $T_{\text{cold}}(f)$ from the initialization to the f 's first LAMBDATA API call of the `get` method. If LAMBDATA has never seen f before, it estimates $T_{\text{cold}}(f)$ by other functions with similar static size as f . Therefore, the time spent in the start phase is

$$T_{\text{start}} = \begin{cases} T_{\text{warm}} & \text{if a warm container is available} \\ T_{\text{cold}}(f) & \text{otherwise} \end{cases}$$

The get phase. Let $D = \{d_1, d_2, \dots, d_n\}$ be the set of data that f needs. We denote by $T_{\text{data}}(d)$ the time to get data d from the cloud storage. We notice that $T_{\text{data}}(d)$ is mainly determined by the size of d and the region of the cloud storage, so LAMBDATA monitors the time spent on recent cloud storage requests to track the relationship between T_{data} and object size dynamically. Let $D_c \subseteq D$ be the subset of cached data and $D_a = D \setminus D_c$ the subset of data absent. If $d_i \in D_c$, then the function can read it immediately. Otherwise, if $d_i \in D_a$, the function needs to get it from the cloud storage, using n concurrent threads (n is an optional annotation provided by the developer, see §4.3.1). In order to calculate the total time to get all the data, we consider two cases. If $|D_a| \leq n$, meaning that there are enough threads to download all data in parallel, then the time is dominated by the slowest thread (i.e., getting the largest data). If $|D_a| > n$, then the time is approximately the total time for getting all data over n threads. Therefore, the time spent in the get phase is

$$T_{\text{get}} = \begin{cases} \max_{d \in D_a} T_{\text{data}}(d) & \text{if } |D_a| \leq n \\ \sum_{d \in D_a} \frac{T_{\text{data}}(d)}{n} & \text{otherwise} \end{cases}$$

Total lead time. As a result, the total lead time is

$$T = T_{\text{start}} + T_{\text{get}}$$

LAMBDATA tries to schedule the function on the Invoker with the smallest T . If the best Invoker is offline or overloaded, then it picks the next one, and so forth. LAMBDATA uses the same load-detection mechanism as existing serverless computing services.

4.4.2 Collecting bookkeeping data

When an Invoker receives a function invocation, it sends an *acknowledgment message* to the Controller. LAMBDATA piggybacks with this message a list of all data that it has currently cached, as an array of (bucket, key, size) pairs. The reason to include data size is to help the scheduling algorithm determine how long it would take to get the data from the cloud storage, on other Invokers that has not cached the data or on the same Invoker if the data is evicted from the cache.

The Controller has a cache registry that maintains a global view of all data currently cached at each Invoker as a dictionary of $\text{Invoker} \rightarrow \text{Set}[(\text{bucket}, \text{key})]$. It also maintains the size of all data as a dictionary of $(\text{bucket}, \text{key}) \rightarrow \text{size}$. Both dictionaries may be stale or incomplete, which only affects scheduling efficiency but not the

correctness of the system. For example, the Controller may think a data is cached on an Invoker, but it has actually been evicted since the last acknowledgment message. In this case, the function simply gets the data from the cloud storage again. If any dictionary grows too large, the Controller just purges old entries.

In order to estimate T_{warm} , T_{cold} , and T_{data} , the Invoker also monitors the time whenever it starts a container or handles a cloud storage operation, and sends them to the Controller via the acknowledgment message. The Controller bookkeeps recent profiling results, and interpolate them for estimation.

4.5 Optimization: Direct File Access

It is common that a function represents data as files, and processes data by reading and writing files. The typical workflow of such a function is first downloading the source file from the cloud storage to local disk, then processing the file to generate an output, and finally uploading the output file to the cloud storage. However, due to container isolations, even if LAMBDATA has already cached the source file, by using the LAMBDATA API, the function still needs to transfer the file from LAMBDATA's cache into its local disk inside the container, and transfer the output file out of the container to LAMBDATA's cache. These copies are the unfortunate overhead because LAMBDATA aims to maintain source code compatibility with the cloud storage service's API.

As an optimization, LAMBDATA allows the function container to mount a portion of the cache as a file system, so that the function can manipulate cached files directly. With the cache mounted, the LAMBDATA API's get method returns a file name instead of the file

content, so the function can directly read the file in the cache. LAMBDATA also promises not to evict the cached file that the function has called `get` on, until the end of the function's life. In addition, the function can also write new files to the cache, and the LAMBDATA API's `put` method takes a file name in place of the argument of value.

LAMBDATA limits the access of files by the function's identity, which is the same access-control mechanism current cloud services employ. Therefore, multiple functions under the same identity can access one another's data, while functions of different identities are isolated.

4.6 Evaluation

We deployed LAMBDATA on Amazon EC2 in the `us-east-1` region, with an `m5a.large` instance as the Controller and five `m5a.2xlarge` instances as Invokers. All instances are running Ubuntu 18.04 and Docker CE 17.03.3. We used Amazon S3 as the cloud storage. We implemented LAMBDATA in Scala 2.12 atop OPENWHISK and wrote all cloud functions in Python 3.6. We limit each function's memory usage to 512MB.

Our experiments aim to answer four research questions:

§4.6.1 Is LAMBDATA fast handling cloud storage requests?

§4.6.2 Does LAMBDATA speed up function invocations?

§4.6.3 Does LAMBDATA speed up multi-function workflows?

§4.6.4 Does LAMBDATA reduce monetary cost?

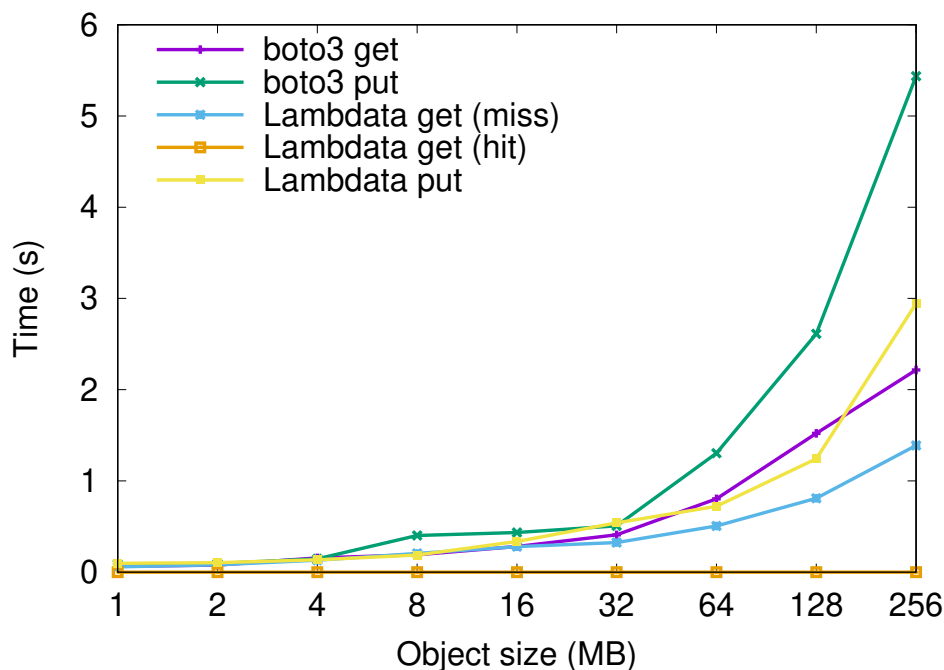


Figure 4.14: Microbenchmark: median time to get and put objects of various sizes to Amazon S3. Lower is better.

4.6.1 Microbenchmark

We first evaluate LAMBDATA’s performance of basic cloud storage operations. We get and put data objects of various sizes, from 1MB to 256MB, to Amazon S3, using both BOTO3, the official AWS Python library, and LAMBDATA. Figure 4.14 shows the median time for each operation.

For the get operation, if the data is cached, LAMBDATA takes less than 1ms, because it is simply accessing files on the local disk, and LAMBDATA does not need to do anything. If the data is missing, LAMBDATA’s performance is comparable to BOTO3’s up to 16MB, and it shows a speedup of up to 1.6× with larger data. For the put operation, LAMBDATA’s performance is comparable to BOTO3’s for small data, and it shows a 1.85× speedup for 256MB data. These speedups are because BOTO3 connects to the cloud storage in the Python run-

time inside a container, whereas LAMBDATA connects to the cloud storage in the Invoker, using a Java runtime outside of containers. Since the LAMBDATA is just a wrapper over the underlying AWS SDK, we do not claim any contribution on the speedups. Nevertheless, the results show that LAMBDATA performs well for basic cloud storage operations.

4.6.2 Function performance

To evaluate LAMBDATA’s performance on running functions, we wrote two serverless applications modeled from real-world applications, each with 10 functions. Table 4.3 lists all functions and shows the parameters used in the experiment. We now briefly describe these two applications.

Photo sharing. We modeled this application according to Instagram, a popular photo-sharing application. Users can upload images, create short video stories from images, apply filters or add special effects, and publish them. The application also includes functions to scan for malware, compress images, and transcode videos. Although Instagram performs many computations (*e.g.*, apply filters) locally on a mobile phone, we implement everything as cloud functions to demonstrate the feasibility. We used images from the Div2K dataset [3, 132] as the workload.

Online classroom. We modeled this application according to Canvas, a popular online learning management system. Teachers can manage lecture notes, assign homework, prepare exams, and grade them. We used documents at one author’s institution as the workload.

Function	Description	Parameters used in this experiment
<i>App 1: Photo sharing</i>		
malware	Scan a file for malware, using yextend [145].	Used malware rules form BinaryAlert [4].
compress	Compress an image, using Pillow [108].	Output JPEG quality = 75%.
thumbnail	Generate a thumbnail of an image, using Pillow.	Thumbnail size = 320×320 .
image_filter	Apply a filter on an image, using Pillow and numpy [101].	Applied an Instagram “Amaro”-like filter.
create_story	Generate a video from a list of images, using OpenCV [102].	1920×1080 M-JPEG, 5 seconds per image.
add_text	Add a text label to a video, using OpenCV.	Added a label with random text.
add_audio	Add an audio track to a video, using FFmpeg [53].	Used a pop music track in MP3 format.
transcoding	Convert a video to another codec, using FFmpeg.	Transcoded into the msmpeg4v2 format.
video_filter	Apply a filter on a video, using OpenCV.	Applied a cartoon-like filter.
publish	Publish an image or a video into a dedicated bucket.	Bookkeeping only, no computation on data.
<i>App 2: Online classroom</i>		
lecture_note	Compile a lecture note in \LaTeX beamer to PDF.	The lecture note had 10 slides.
merge_notes	Merge a list of PDF files into one PDF.	Merged 10 lecture notes.
watermark	Add a watermark to all pages of a PDF file.	Used “lecture notes” as the watermark.
split_note	Split a PDF file into two files (slides and speaker notes).	Split a 10-page PDF into two 5-page PDFs.
write_homework	Compile a \LaTeX homework document to PDF.	The PDF had three questions, one per page.
grade_homework	Generate per-question PDF files for all submissions.	Used three questions, 20 student submissions.
question_pool	Create a question pool for an exam from a PDF repository.	Randomly chose 5 out of 15 questions.
make_exam	Generate per-student problem set from the question pool.	Randomly chose 3 out of 5 questions.
answer_exam	Compile a \LaTeX document and attach it to the exam PDF.	The document had 10 pages.
grade_exam	Attach a grade on each page of a PDF file.	The document had 10 pages.

Table 4.3: List of all functions and the parameters used in the experiment.

Speedup of function invocations

To evaluate the speedup of function invocations, we instrumented the time spent on each phase of a function invocation. Table 4.4 shows the breakdown of each phase’s time, and the speedup of LAMBDATA compared with OPENWHISK, where all numbers are the median of 100 invocations. In order to eliminate the variance in the start time and make a fair comparison, we pre-warmed the container before each invocation,

First, we find that cloud functions are small and short-running. With practical workloads, all function invocation times are shorter than 10 seconds, and the majority shorter than 2 seconds.

We further find that the time spent on each phase varies significantly across functions. Both get and put phases take hundreds of milliseconds for most functions. They are mainly determined by the number of data objects and each object’s size. For example, create_story and merge_notes have longer get time because they need to get multiple files from the cloud storage; video_filter has longer put time because the size of a video file is large. The times of the compute phase are diverse, ranging from 0 to 8 seconds. They are mainly determined by the function’s business logic. For example, the publish function only performs some bookkeeping and does not modify the data, so it spends less than a millisecond in the compute phase. On the other hand, image_filter and video_filter perform heavy numerical computations on the data, resulting in about 8 seconds.²

We compared the run time of LAMBDATA with OPENWHISK and found no statistically

²We implemented both functions in Python. The time could be shorter if it used native code or GPU.

Function	get	compute	put	speedup
<i>App 1: Photo sharing</i>				
malware	208	69	0	3.99×
compress	216	117	83	2.08×
thumbnail	201	79	48	2.58×
image_filter	130	8129	111	1.02×
create_story	651	865	418	1.51×
add_text	353	938	392	1.27×
add_audio	419	70	249	2.31×
transcoding	414	591	154	1.56×
video_filter	398	7600	461	1.05×
publish	299	0	423	1.71×
<i>App 2: Online classroom</i>				
lecture_note	116	1345	122	1.08×
merge_notes	615	1436	145	1.39×
watermark	129	513	116	1.20×
split_note	65	89	157	1.27×
write_homework	98	1470	123	1.06×
grade_homework	711	904	311	1.58×
question_pool	101	83	102	1.54×
make_exam	129	61	114	1.74×
answer_exam	115	1340	143	1.08×
grade_exam	133	856	139	1.13×
Geometric mean				1.50×

Table 4.4: *Breakdown of the phases in each function.* All times are in milliseconds. The last column shows the speedup with cached data.

significant difference if the data is not in the cache. If the data is cached, LAMBDATA gives an average speedup of 1.50×. Of all functions, the speedup is higher if the get phase dominates, and lower if the compute phase dominates. For example, malware has 3.99× speedup because its computation is fast, and LAMBDATA’s cache eliminates its need to get data from the cloud storage. On the other hand, the speedup of image_filter is only 1.02× because it spends most of its time doing the computation, and the cache helps little.

Overall, our experiment shows that LAMBDATA offers significant speedup over existing serverless clouds.

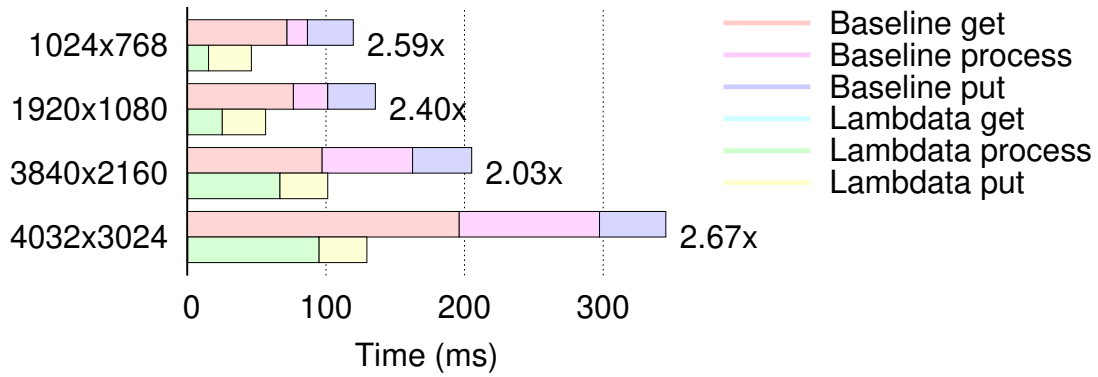


Figure 4.15: *Time breakdown and speedup of the thumbnail function for various image sizes.*

Case study

In order to understand how data matters to LAMBDATA’s performance, we show a case study on thumbnail, a typical cloud function. This function gets an image file, generates its thumbnail, and puts it on the cloud storage. We study how various image size affects both OPENWHISK and LAMBDATA’s run time in each phase.

We pre-warmed the container and ran thumbnail with input images of four popular dimensions: 1024×768 (web quality, 460KB), 1920×1080 (full HD, 1.2MB), 3840×2160 (4K UHD, 4.7MB), and 4032×3024 (12 megapixel iPhone photo, 7MB). Figure 4.15 shows the timeline of each function invocation. Each cluster represents an image dimension, of which the top bar is the timeline of OPENWHISK, and the bottom bar is the timeline of LAMBDATA.

We observe that all phases’ run time increase with the image dimension, but the ratio of these increases is non-linear. For example, the compute time on an iPhone image is 6.9 times as long as on a web image, while the get time is only 2.6 times as long. LAMBDATA shortens the get time to almost zero, but the compute time remains the same. Therefore,

Workflow	Description	Functions
<i>App 1: Photo sharing Slideshow</i>	The user creates a short 1080P M-JPEG video with five recently-uploaded images, and then adds some text to the video. The application then transcodes the video to the msmpeg4v2 format and publishes the converted video.	create_story add_text transcoding publish
<i>App 2: Online classroom Distribute handouts</i>	The user compiles 10 lecture notes from \LaTeX beamer source codes to PDFs. When all compilations finish, the application merges all lecture notes into one PDF file, applies a watermark on each page, and splits the notes into two PDF files: one with all slides, the other with speaker notes.	lecture_note merge_notes watermark split_note

Table 4.5: *Description of two representative workflows.*

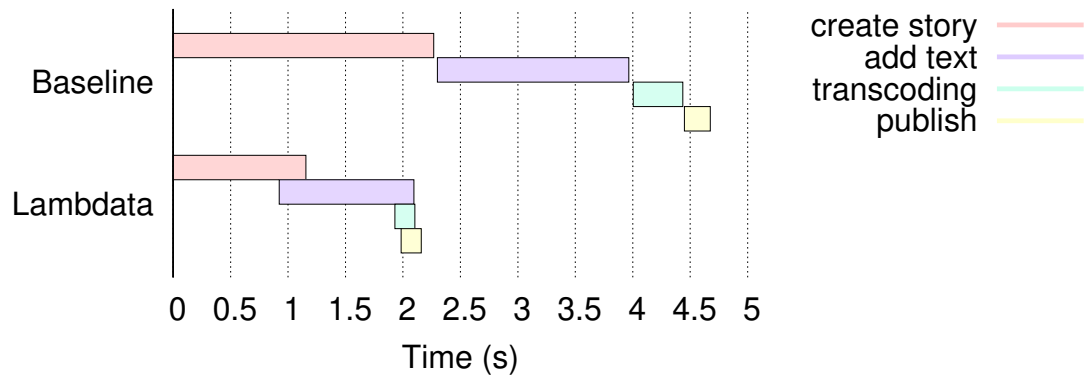
the speedup LAMBDATA provides is non-linear with regard to the image dimension. We find that while LAMBDATA gives speedup for all images, it works best with iPhone photos.

The same observation applies for all functions. We find that LAMBDATA works well with all practical real-world data.

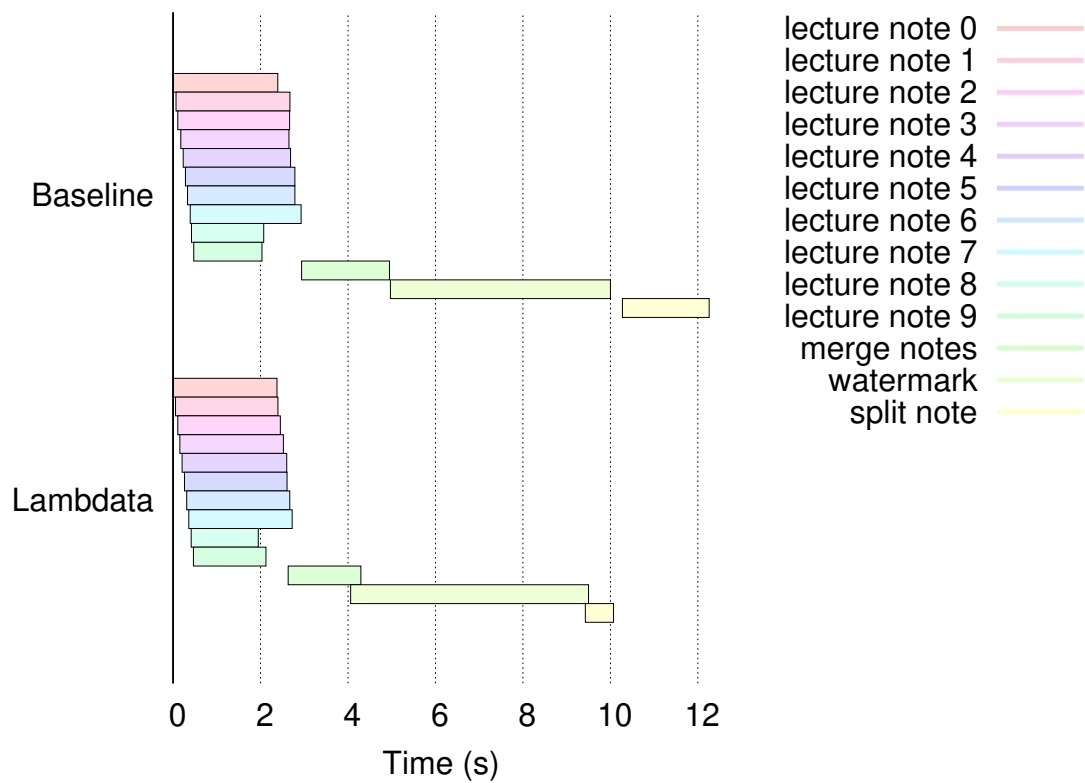
4.6.3 Workflow performance

To evaluate LAMBDATA’s performance on real-world usage involving multiple functions, we simulated 10 workflows that resemble practical scenarios and used synthetic workloads derived from real-world parameters to trigger these functions. We show case studies of the two most representative workflows, and the rest is similar. Table 4.5 lists these workflows. We ran each workload 15 times and chose the result with the median turnaround time.

Figure 4.16 shows the timeline of each workflow. In each graph, the top cluster is the baseline running on OPENWHISK, and the bottom cluster is of LAMBDATA.



(a) Slideshow



(b) Distribute handouts

Figure 4.16: *Timeline for two representative workloads.*

Slideshow. This workflow has four functions chaining into one pipeline. Since the user has just uploaded the source images, the data cache is warm. For each individual function, LAMBDATA's run time is shorter than the baseline, because it reuses data from the cache. The difference is most significant with `create_story` and less so with `add_text`, for the

same reason described in §4.6.2.

For the orchestration of the four functions, the baseline schedules each function invocations consecutively, whereas LAMBDATA overlaps the next-stage functions with put phase of the previous function, further reducing the turnaround time. We notice that at around the 2s time mark, three functions overlap, possible because transcoding's computation is so fast that `add_text` has not finished its put phase yet.

As a result, LAMBDATA reduces both the time in each individual function because of caching, and the turnaround time because of overlapping. Overall, LAMBDATA achieves a $2.16\times$ speedup to finish the workflow.

Distribute handouts. In the first stage of this workflow, the user invokes 10 concurrent functions computing distinct files. Because our Invoker machines have 8 cores, both the baseline and LAMBDATA effectively schedule 8 invocations on Invoker1 and the rest on Invoker2, which is why the first 8 invocations take slightly longer time than the last two in the timeline.

In the second stage, `merge_notes` gathers the 10 output data objects and merges them together. OPENWHISK schedules this invocation on Invoker3, whereas LAMBDATA schedules it on Invoker1. Therefore, LAMBDATA reuses 8 of the 10 data objects from the cache and gets the remaining two objects from the cloud storage.

The remainder of this workflow forms a pipeline, and LAMBDATA both reuses cached data and overlaps functions, similar to the slideshow workflow. As a result, LAMBDATA achieves a $1.22\times$ speedup to finish the workflow.

All workflows. Overall, LAMBDATA achieves an average of $1.51\times$ speedup across all

Workflow	Baseline	LAMBDATA	Savings
<i>Workflow 1: slideshow</i>			
Serverless cost	40.8	24.1	
Storage cost	21.6	20.4	
Total	62.4	44.5	28.6%
<i>Workflow 2: distribute handouts</i>			
Serverless cost	276.8	256.0	
Storage cost	73.8	65.8	
Total	350.6	321.7	8.2%

Table 4.6: *Monetary cost*. Numbers are in $\times 10^{-6}$ dollars.

workflows.

4.6.4 Cost savings

LAMBDATA’s cost savings come from two factors. First, LAMBDATA shortens the run time of cloud functions, thus reducing cost on the serverless computing service. Second, LAMBDATA eliminates redundant requests to the cloud storage, thus reducing cost on the storage service. We applied the current pricing model of AWS Lambda and AWS S3 and calculated the cost savings for all workflows.³

Table 4.6 shows the cost savings for the two workflows in §4.6.3, where LAMBDATA reduces their cost by 28.6% and 8.2%. Across all workflows, LAMBDATA’s achieves an average cost savings of 16.5%.

4.7 Discussion

We now discuss some design implications of LAMBDATA.

³We did not use AWS’s billing statement because it was too coarse-grained.

Cache coherence. LAMBDATA requires that data are immutable, so the cache is never incoherent. This requirement follows our insights (§4.2.1) and is the best practice of writing serverless applications. If a function needs to mutate data, it should store the data with a new key and delete the old data.

Concurrent writes. Writing different data to the same object from multiple functions violates our requirement that data are immutable, and is a bad practice in any serverless computing. LAMBDATA does not prevent concurrent writes but leaves it to the developer to use distinct object keys.

Cache eviction policy. The cache eviction policy is orthogonal to LAMBDATA’s design. LAMBDATA can use any policy to evict cache. Because data in serverless computing are small and demonstrate good temporal and spatial locality, the cache need not be large. In practice, we found a simple LRU algorithm works well.

Data prefetching. By making a cloud function’s data intents explicit, one further optimization is that the Invoker can prefetch data on behalf of a function, while the container is being initialized. LAMBDATA did not implement this optimization, because cloud providers charge users by the duration a container runs, and this prefetching happens outside a container’s lifetime, thus complicating the billing model.

Security. LAMBDATA maintains the same container isolations as in existing serverless clouds, except for the cache. It leverages existing cloud services’ identity and access management (IAM) policies to restrict what cache a function can access. Only functions under the same identity can see one another’s cached data. A malicious function could try to

cache a lot of data in the hope of exhausting the cache space and evicting other identities' cached data. LAMBDATA can mitigate this impact by imposing a limit on the maximum cache size per identity.

4.8 Related Work

LAMBDATA builds upon prior work that we now describe.

Memoization for dataflow programs. Memoization [87, 114, 93] is a technique that reuses prior computation results of pure functions. Nectar [69] manages data and computation in the traditional data center setting. It memoizes intermediate computation results of Dryad programs. Incoop [21] uses memoization on the MapReduce framework. However, these systems only work for specific programming models, and it is non-trivial to generalize their use cases to the serverless computing setting. LAMBDATA generalizes the idea of memoization to cloud functions.

Scheduling workflows. Workflow execution engines coordinate multiple tasks. For example, Oozie [79] manages workflows for Hadoop systems, and Yu *et al.* [148] schedules workflows for grid computing. Unfortunately, their models do not fit serverless programming.

Serverless function orchestration. AWS Step Functions [123] manages serverless computing workflows by describing functions as a state machine. Although it maintains states for serverless applications, the 32KB size limit is insufficient for large data objects. IBM Composer [74] and Azure Durable Functions [94] let developers write function compo-

sitions with special library functions, and allow larger state size. Nevertheless, these frameworks do not consider data locality, and their states are only for intermediate data, not persisted in the cloud storage. Besides, they all require new programming models unfamiliar to developers. By contrast, with LAMBDATA, developers write functions and manipulate data objects in a familiar way.

Data systems for serverless computing. Pocket [82] introduces a multi-tier storage system for interactive serverless data-analytic applications. However, it focuses on intermediate data and does not provide high data durability. By contrast, we choose to build LAMBDATA on top of existing cloud storage so that data are highly durable, and developers are familiar with this programming model.

4.9 Summary

This chapter presented LAMBDATA, a novel serverless computing system that enables developers to declare a cloud function’s data intents, including both data read and data written. It caches data locally, and its data-aware scheduling algorithm considers both code and data locality. Evaluation on two practical applications with 20 cloud functions shows that it achieves an average of $1.51\times$ speedup on the turnaround time and reduces monetary cost by 16.5%. The source code of LAMBDATA and the applications used for evaluations are at <http://columbia.github.io/lambdata>.

Chapter 5

Limiting Mobile Data Exposure with Cloud-based Idle Eviction

The final paradigm shift in the era of cloud computing happens in mobility. Mobile technology is replacing desktops as the primary personal computing platform and is being used in increasingly sensitive contexts. For example, today’s users rely on smartphones and tablets to access their personal and corporate email, prepare tax returns, and review customer documents [110]. Even the US military recently announced that it will equip soldiers with Android devices for accessing classified documents [97]. The draw to new mobile technology is justifiable: mobile devices offer convenient and constant connectivity, increase productivity, and provide access to feature-rich, cloud-based applications (a.k.a. “apps”).

Despite these advantages, the transition to mobile devices raises serious and yet unresolved concerns, particularly with respect to data security in the event of device theft and loss. Unlike desktops, generally assumed to be physically secure, mobile devices are extremely prone to theft and loss. Statistics here are staggering: 49% of the New York City population has experienced mobile phone loss/theft [88], and the FCC recently declared mobile theft an “epidemic” in major US cities [52].

Though alarming, these statistics have yet to prompt mobile OSes to address the serious data-security threats posed by device theft or loss. Like their desktop precursors, such

as Linux and Mac OS X, mobile OSes let sensitive data accumulate uncontrollably on the device. For example, the OS accumulates significant amounts of data in cleartext memory, and the file system retains deleted files by not purging their contents. Despite being backed by clouds, applications hoard sensitive data – such as emails, documents, and banking information – on the vulnerable device. Although encrypted file systems [96], encrypted RAM [112], and remote-wipeout systems [12, 77] help protect this data, they are imperfect stopgaps for OSes that were simply not designed with physical insecurity in mind. For example, a recent study shows that 57% of corporate users employ no locking mechanisms on their smartphones, rendering encryption useless [110].

This chapter presents CLEANOS, a new Android-based mobile operating system¹ designed to *manage sensitive data rigorously and maintain a clean environment at all times in anticipation of device theft*. The crucial insight in CLEANOS is to leverage the tight integration of today’s mobile applications with trusted cloud-based services in order to evict sensitive in-memory and on-disk data to those services whenever it is not needed on the device. CLEANOS thus ensures that the minimal amount of sensitive data is exposed on the vulnerable device at any time.

CLEANOS extends Android in two major ways. First, it introduces *sensitive data objects* (SDOs), a new abstraction that facilitates management of sensitive data on mobile devices. An SDO is a logical collection of Java objects, files, and database items that applications create and use to manage their sensitive data, such as emails, financial data, or documents. SDOs and their data “disappear” from the device unless they are frequently

¹We view the OS notion broadly in this dissertation to include both the traditional OS and the entire Android framework on which apps run.

used by an application. For example, if an email app adds an email’s content to an SDO, any “trace” of that content automatically disappears from RAM and stable storage unless the user is actively reading that email on an unlocked screen. Recovering the email requires interaction with the cloud.

Second, to evict idle SDOs, CLEANOS modifies Android’s Java interpreter (Dalvik) to introduce a new type of Java garbage collector (GC), called an *evict-idle GC* (eiGC). While a traditional GC deallocates only those objects guaranteed to never be used in the future (i.e., no pointers to them exist), eiGC eliminates objects that have not been used for a period of time *even if* they might be used again in the future (i.e., pointers to them still exist). To do so, eiGC walks through all Java objects in an idle SDO and encrypts their data-bearing fields, such as primitives and arrays of primitives, with a key that is escrowed in the cloud. Our modified Dalvik interpreter then faults when a bytecode instruction executes on an evicted object, retrieves the key from the cloud, and decrypts the object. Thus, data eviction in CLEANOS is logical; the data itself remains on the device in encrypted form, while the key is shipped to the cloud.

The major security benefit of CLEANOS stems from the value-added services that app clouds can build on top of it. For example, a cloud could revoke data access following a theft report, provide an audit log of data exposed upon theft, or monitor data access to detect anomalous uses. Building such services on today’s “dirty” devices would be tremendously challenging and likely require sacrificing semantics or performance. For example, Gmail allows email access revocation [67], but emails cached on the device remain exposed. Conversely, not caching sensitive data on the device degrades performance over slow mobile networks. CLEANOS provides device-side OS support for building robust,

Component	New or changed features
Dalvik (JVM)	Evict-idle Garbage Collector (eiGC) Eviction-aware bytecode interpretation Secure deallocation of interpreted stacks
Android SDK	SDO API Default SDO heuristics
TaintDroid	Support for millions of taints SQLite vulnerability fix
SQLite	Taint tracking in database
Webkit	Screen-buffer purging
Bionic (libc)	Secure user-space deallocation
Linux kernel	Secure page deallocation with <code>grsecurity</code>

Table 5.1: *CLEANOS Modifications to Android, TaintDroid.*

secure, and efficient value-added cloud services.

We built CLEANOS in Android using the TaintDroid taint-tracking system [49] and also implemented a value-added cloud service that provides post-theft data-exposure auditing. To do so, we modified several core components in Android and TaintDroid, summarized in Table 5.1. Together, our changes provide: (1) eviction of idle Java objects, (2) heuristics for identifying sensitive data without requiring app changes, (3) support for millions of taints in TaintDroid, and (4) multi-layer secure deallocation of freed data in Java, native, and kernel space. While CLEANOS’s design extends in-memory eviction to stable storage, this dissertation and our current prototype focus on in-memory data eviction.

Overall, we make the following contributions:

1. We demonstrate the sensitive data exposure problem by analyzing 14 popular Android apps.
2. We define SDOs, a new abstraction for managing sensitive data on theft-prone de-

vices.

3. We implement CLEANOS, an Android OS extension that combines known encryption-based data destruction [23, 61, 106] with a new GC process that evicts idle sensitive data.
4. We present a set of valuable add-on services that clouds could build on top of CLEANOS.

The remainder of this chapter is organized as follows. The next section presents a case study on the data exposure issue. §5.2 describes CLEANOS’s goals and assumptions. §5.3 shows the architecture. §5.4 describes the implementation. §5.5 demonstrates CLEANOS’s applications. §5.6 shows evaluation results. §5.7 discusses some security implications. §5.8 presents related work, and §5.9 summarizes this chapter.

5.1 Case Study: Data Exposure on Android

We selected for analysis 14 Android apps according to their popularity in five sensitive categories: email, finance, document editing, password management, and social networking. We define as *exposed* any data that persists on the device – either in RAM or on storage – for a prolonged period of time, such as 10 minutes (§5.2 describes our rationale for this threat model). Our goal in the analysis was to answer three questions: (1) Is sensitive-data exposure a real problem? (2) If so, what are its causes? and (3) Is the exposure necessary? We tackle each question using examples from our analysis.

Is Data Exposure a Real Problem? We installed the 14 apps on a rooted Nexus S phone

App	Extracted cleartext data
Email	Password, email contents, subjects, from/to, contacts
OI Notepad (doc)	Document and metadata
KeePass (password manager)	App password, all stored passwords & descriptions
Pageonce (finance)	Password, transactions, bank account information
Facebook (social)	Wall posts and messages

Table 5.2: *Examples of captured sensitive data.*

with Android 2.3.4 and asked the following question: what kinds of sensitive data can one find by dumping RAM and database contents while apps run in the background? Our acquisition process was vastly simplified by our rooted phone and the lack of encryption on the default Android configuration. Nevertheless, we believe that our findings indicate the level of data exposure on better-protected phones in face of realistic, albeit sophisticated, attacks, such as cold boot RAM imaging [70]. We created a stable-state environment – akin to the one a thief might find on a lost device – by ensuring that apps had not been used for 10 minutes prior to taking RAM and DB dumps.

The answer to our question is eye-opening: with simple techniques, we retrieved cleartext copies of sensitive information from *all but one app*. Table 5.2 shows examples of cleartext sensitive data we extracted from a select subset of the apps. Table 5.3, column “*Extracted Cleartext Data*,” expands the result set to all 14 apps and categorizes data in three classes of varied sensitivity: passwords, contents (e.g., email body, document content, bank account), and metadata (e.g., email subject, document title). Overall, we captured passwords in 5/14 apps, contents in 11/14 apps, and metadata in 13/14 apps.

What Causes Data Exposure? Given these results, an obvious question is what leads to so much leakage. There are several possible answers:

App	Description	Extracted			Cleartext data hoarding					
		cleartext data			RAM			SQLite DB		
		Pass word	Cont ents	Meta data	Pass word	Cont ents	Meta data	Pass word	Cont ents	Meta data
Email	Email (default)	Y	Y	Y	Y	-	Y	Y	Y	Y
Gmail	Email	-	Y	Y	-	-	-	-	Y	Y
Y! Mail	Email	Y	Y	Y	-	-	-	-	Y	Y
GDocs	Documents	-	-	Y	-	-	Y	-	Y	Y
OI Notepad	Documents	-	Y	Y	-	Y	Y	-	-	-
Dropbox	Documents	-	-	Y	-	-	Y	-	-	Y
KeePass	Password mgr	Y	Y	Y	Y	Y	Y	-	-	-
Keeper	Password mgr	Y	Y	Y	Y	-	Y	-	-	-
Amazon	Commerce	-	-	-	-	-	-	-	-	-
Pageonce	Finance	Y	Y	Y	Y	Y	Y	-	-	Y
Mint	Finance	-	Y	Y	-	Y	Y	-	Y	Y
Google+	Social	-	Y	Y	-	-	-	-	Y	Y
Facebook	Social	-	Y	Y	-	-	-	-	Y	Y
LinkedIn	Social	-	Y	Y	-	-	Y	-	Y	Y

Table 5.3: *Exposure of cleartext sensitive data across all 14 apps.* A ‘Y’ indicates that we obtained cleartext copies from RAM/DB. A ‘-’ does not mean that the data is not on the device, but just that we did not find it in cleartext; the data could exist in some encrypted form.

Insecure Deletion: The Android OS, including the kernel, system libraries, and the Java framework, leaks sensitive information by not erasing data securely after it is deallocated or by not securely erasing files when an app asks it to do so. These problems are well known in desktop and server settings and have been addressed with secure deallocation [29] and assured deletion [106, 131], respectively.

OS Data Buffering: Recent work shows that OSes and device drivers retain data in buffers past its intended life. It also shows how to limit OS-buffered data exposure [45].

App Data Hoarding: Although most of the apps are cloud-based, our experiments show that they hoard significant amounts of cleartext sensitive information on the device, either

App	Data	When app uses data
Email	Password	User/automatic refresh
	Subjects	On the email list screen
	Contents	User opens the email
OI Notepad	Note title	On the note list screen
	Note body	User edits the note
KeePass	Master password	App launches
	Entry name	On the entry list screen
	Entry password	User opens the entry

Table 5.4: *Examples of when hoarded sensitive data is being actually used by the apps.*

in RAM or in the local database. For example, the default Android email app maintains the email account password in cleartext RAM *at all times*, while KeePass, a popular password manager, loads its entire password database into RAM at startup and keeps it there. Column “*Cleartext Data Hoarding*” in Table 5.3 shows the *persistent*, app-intended cleartext data we found in RAM or DBs.² It demonstrates that the hoarding behavior is pervasive: all but one of the 14 apps permanently maintain at least one type of sensitive data either in RAM or in the database, while 6/14 apps permanently maintain their passwords or some sensitive content in RAM.

Memory Leaks: Beyond the scope of our experiments is the well-known ease of unwittingly introducing memory leaks into Android applications [11]. If small, these leaks may go undetected and expose sensitive information.

Is Data Exposure Necessary? Although apps hoard significant amounts of sensitive data on mobile devices, they tend to access this data fairly infrequently, suggesting that

²For RAM, we conservatively assume an object to be persistent if it always appears in the app’s Java object dump.

data is often exposed longer than it needs to be. By way of example, Table 5.4 identifies situations where three of our most problematic apps use hoarded sensitive data. For example, the password in the default Android Email app, which we know is exposed in RAM at all times, is in fact used only during inbox refreshes (the default is every 15 minutes). Similarly, each email’s content is exposed in SQLite at all times but accessed only when the user opens that particular email. While the frequency of these operations depends on the workload, intuitively it should be relatively rare, making prolonged exposure unnecessary.

Implications for Mobile OS Design. Secure deletion for storage, RAM, and OS buffers has been acknowledged as, and developed into, a primary OS function [29, 106, 45]; however, the management of app-driven data hoarding or leakage has thus far been considered an app’s own responsibility. For example, faced with similar data-hoarding practices in desktop and server applications, Chow, et al. [29] conclude that “little can be done without modifying the application” and that “leaks are recognized as bugs by application programmers, so they are actively sought after and fixed.” Unfortunately, relying on the app to manage sensitive data is problematic. Sensitive-data caching presents tradeoffs between security on one hand and performance, usability, and energy/bandwidth consumption on the other hand. Without solid abstractions, calibrating these tradeoffs is challenging. For example, should a document-editing app cache the documents locally for good performance over cellular networks (as recommended in some mobile app guidelines [59]), or should it not do so for security reasons (as recommended in other guidelines [136])? Should it cache the user’s password for convenience, or should it prompt the user for it

whenever it is needed?

We argue that mobile OSes *can* and *should* offer abstractions for apps to manage their sensitive data rigorously *without* sacrificing their performance, usability, or other properties. This dissertation introduces one such abstraction in CleanOS, whose goals we next describe.

5.2 Goals and Assumptions

Goals. The primary goal of CleanOS is to minimize the exposure of an app’s *allocated* sensitive data by evicting it from the device whenever the data is idle (i.e., not being actively used by the application). The key insight that makes this possible is the tight integration between today’s mobile apps and cloud services. CleanOS leverages clouds to create a new abstraction, called a *sensitive data object* (SDO). SDOs track sensitive information as it flows through RAM and stable storage. As soon as they become idle, they are automatically evicted to the cloud and are recovered only when the app needs them again.

Specific design goals of CleanOS include:

1. *Eviction:* SDOs should “disappear” as soon as they become idle whether or not they are expected to be used by an application in the future.
2. *Reasonable performance:* We seek to provide reasonable performance for popular mobile apps despite data eviction over Wi-Fi or cellular networks.
3. *Reasonable defaults:* While we admit app changes for best performance and semantics, we aim to offer reasonable defaults even for unmodified apps.

4. *Leverage technology trends*: CleanOS must integrate naturally with existing tech trends, such as the tight integration of mobile apps with cloud services.
5. *Design for mobiles*: CleanOS' design should target mainstream mobile technologies, such as Android.

Eviction of idle data (Goal 1) is our primary goal and contribution in CleanOS. We strive to ensure that a thief cannot get a “free lunch” by capturing a device. Rather, he should be required to contact the cloud in order to access data of interest, at which time the cloud could deny access, log it, rate-limit it, etc. However, enforcement of precise timeouts on idle sensitive data is a non-goal. From a performance perspective (Goal 2), we wish to ensure that popular apps remain usable despite eviction across Wi-Fi or cellular networks (e.g., 3G/4G).

A common pitfall when proposing new OS abstractions is to require application changes to gain any benefit. To avoid this, CleanOS should include heuristics to construct default SDOs that provide reasonable eviction and performance properties even for unmodified apps (Goal 3). Finally, we aim to exploit unique properties of popular mobile technologies in CleanOS' design (Goals 4 and 5). First, we leverage the tight integration between most mobile apps and trusted cloud services to evict device data to those services. For local-only apps, however, the user can still integrate them with his own CleanOS service. Second, while the data eviction concept is applicable to any mobile OS, we focus our design on Android, which lets us leverage its technological properties to facilitate data eviction. For example, since all Android apps are written in Java, we decided to tap into the garbage collector to evict idle sensitive Java objects.

Threat Model. Our threat model considers *any data on a mobile device to be vulnerable to data-driven thieves*. While many data protection systems exist – including encrypted file systems [96, 129, 48], encrypted RAM [112, 86, 107], and data wipeout systems [12, 77] – they are imperfect when confronted with negligent users or (sophisticated) physical attacks. First, users can foil any protection system by not locking their devices [110], assigning trivial PINs or passwords [75], or writing passwords down in easily retrievable locations [120]. Second, mobile devices are prone to physical attacks, which are notoriously difficult to protect against. For example, an attacker could use cold boot attacks [70] to retrieve in-RAM decryption keys or data, break the seal of tamper-resistant hardware [10, 117], or shield the device from the network to prevent remote wipeout [12]. Such threats are especially relevant for corporate, government, and military users, who interact with particularly sensitive data, such as trade secrets, customer data, health data, or state secrets.

To maintain post-loss control over data despite such threats, CleanOS evicts data to a cloud service, which is assumed to be trusted and non-compromisable. In reality, mobile users are already required to trust the clouds on which their apps rely, so our assumption is reasonable. Depending on the deployment model, these clouds could integrate directly into CleanOS to help cleanse their apps automatically. For apps without a cloud component, we assume that users can evict data to a trusted community or self-administered CleanOS service. Finally, we assume that the cloud learns about a monitored device’s theft, either directly from the user or via an automatic mobile-theft detection mechanism.

CleanOS explicitly assumes that the mobile device, along with all software running on it, is trusted until it is lost. For example, the thief cannot install malware on a user’s

device, tamper with the device physically, or inspect it prior to stealing the device. After loss, we trust neither the hardware nor the software on the device.

We assume that disconnection is the exception rather than the rule. With pervasive wireless and cellular network coverage, this assumption is becoming increasingly realistic. Moreover, CleanOS is especially geared toward cloud-based apps, which typically require connectivity for full functionality. Nevertheless, we present techniques to allow disconnected operation in certain cases.

CleanOS is most applicable to long-lived daemon-like apps, whose execution consists of brief computation sessions interspersed with long periods of inactivity. Most of today’s mobile apps follow this model, including email, browsers, document editors, and social apps. CleanOS disables exposure during periods of inactivity.

Finally, we explicitly assume the existence of robust secure deallocation and OS buffer-cleanup techniques [29, 106, 45] and do not aim to improve the state of the art in these intensely-researched directions. Rather, we focus on limiting the exposure of sensitive data that *applications* hoard or leak, a problem previously thought intractable from an OS perspective (see §5.1).

5.3 The CleanOS Architecture

We now describe our CleanOS design for Android. We focus initially on in-RAM data eviction, after which we show how to extend SDOs to stable storage.

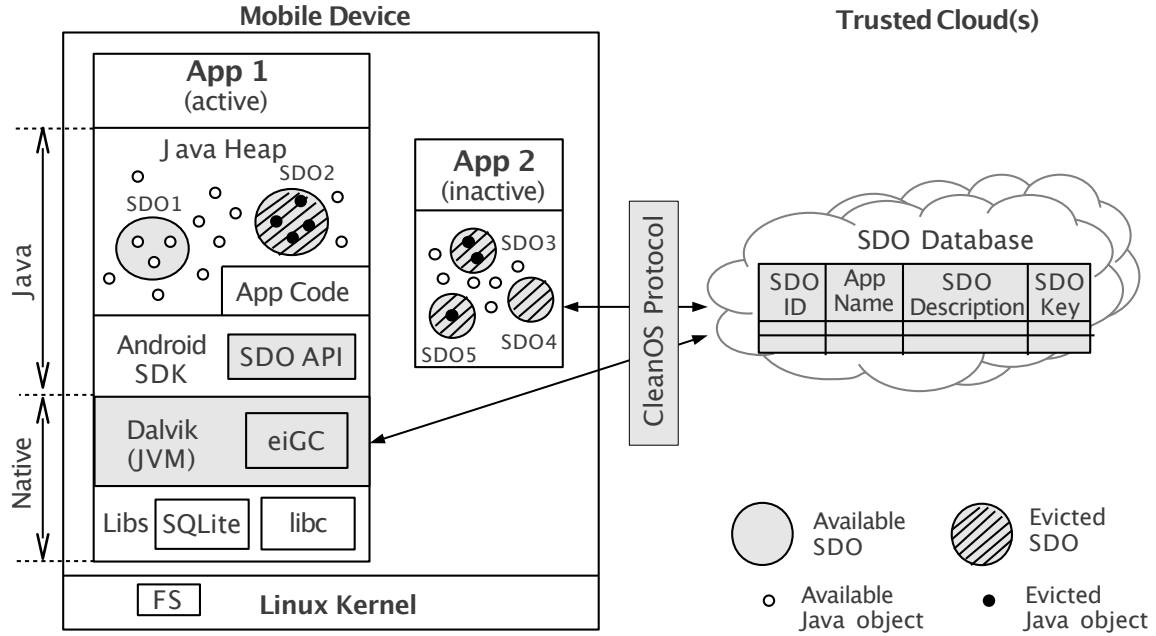


Figure 5.1: *The CleanOS Architecture*. Key components are highlighted in grey. We add or modify in some way all of the boxed components (except for FS and kernel).

5.3.1 CleanOS Overview

Figure 5.1 shows the CleanOS architecture, which includes three major components: (1) the *sensitive data object* (SDO) abstraction, (2) a modified, eviction-aware version of the Dalvik interpreter, along with an *evict-idle garbage collector* (eiGC), and (3) the SDO cloud store. Briefly, apps create SDOs and place their sensitive Java objects in them. The modified Dalvik tracks their propagation across RAM with TaintDroid and monitors their bytecode-level accesses. The eiGC evicts SDOs to the cloud if they remain idle for a specified period.

An SDO is a logical collection of Java objects, such as string objects representing the emails in a thread or objects pertaining to a bank account in a finance app. Upon creation, SDOs are assigned app-wide unique IDs and encryption keys (K_{SDO}), and are registered

with the cloud.

We implement three functions for SDOs. First, we track objects in an SDO with a modified TaintDroid system, using the ID as a taint. As objects are tainted with an SDO's ID, they become part of the SDO. For example, SD01 in Figure 5.1 includes three objects added to it either explicitly by the app or automatically by our modified Android framework. Second, we monitor accesses to SDOs and record their timings. Whenever an app accesses an object in an SDO (e.g., to compute on it, send it over the network, or display it on screen), that SDO is marked as used. Third, we evict SDOs when they are idle for a time period (e.g., one minute).

To evict idle SDOs, the eiGC eliminates unused Java objects from RAM *even if* they are still reachable. It periodically sweeps through Java objects and evicts them if they are tainted with an idle SDO's ID. In Figure 5.1, the active app (App 1) has one available SDO (SD01) and one evicted SDO (SD02). For example, an SDO associated with an email thread might be available while the user reads emails in that thread, but the password SDO might remain evicted. When the app goes into the background, all of its SDOs might be evicted, as shown for App 2. An SDO is evicted when all Java objects in it have been evicted; however, an available SDO may have both evicted and available Java objects.

Conceptually, eviction occurs at the level of logical SDOs. In practice, however, CleanOS must eliminate the actual data-bearing objects from the vulnerable device. To do so, eiGC leverages encryption-based data destruction from assured-delete file systems [106, 61] and applies it to the memory subsystem. Specifically, eiGC replaces data-bearing fields in objects, such as primitives and arrays of primitives, with encrypted versions and then securely destroys the encryption key. To encrypt a data field F , eiGC uses

a key K_F that is uniquely generated from the SDO's key K_{SDO} in the cloud (see §5.4 for details). We modified Dalvik to fault when an app attempts to access the evicted data, at which time it retrieves K_{SDO} from the cloud, generates K_F , and decrypts the data. K_{SDO} is then cached onto the device and securely removed when the SDO as a whole is again evicted.

We next provide more detail on the two main contributions of CleanOS: the SDO abstraction and the eiGC.

5.3.2 The SDO Abstraction

SDOs fulfill two functions in CleanOS. First, they let CleanOS identify sensitive data and focus its cleansing on that data for improved performance. Indeed, evicting all Java objects indiscriminately would be prohibitively expensive, while evicting a few at random would diminish security benefits. Second, SDOs are instrumental in supporting some of our envisioned add-on cloud services, such as the auditing service described in §5.5, as they identify and classify sensitive data for the auditor.

APIs. Figure 5.2 shows the SDO API. To realize the data-control benefits of CleanOS, apps create SDOs and add/remove Java objects to/from them. To create an SDO, an app specifies a description, which is a short, human-readable string that describes the sensitive data associated with the SDO. For example, our modified email app, CleanEmail, creates an SDO using “*password*” for the description and adds the password object to it. It also creates one SDO for each email thread, specifying the thread's subject as the description, and adds each email in the thread to it. Section 5.5 describes two apps that we trivially

SDO API:

```
class SDO {  
    SDO(String description, SDOLevel level) // new SDO  
    void add(Object o)           // adds object to SDO  
    void remove(Object o)       // removes object from SDO  
}
```

CleanOS Protocol:

```
registerSDO(sdoID, appName, description, key)  
    // registers SDO with DB  
fetchKey(appName, sdoID, bucketID) → key || null  
    // fetches the key for a bucket in the SDO  
    // bucketID = 0 returns the SDO's key  
sdoEvicted(appName, sdoID)  
    // announces an SDO's eviction to the cloud
```

Figure 5.2: *The CLEANOS SDO API and device-cloud protocol.*

ported to CleanOS with minimal modifications (fewer than 10 LoC).

Figure 5.2 also shows the protocol used to register SDOs, retrieve their keys after eviction, and report their eviction to the cloud. To create an SDO, the app registers it with the cloud database using the SDO API, specifying its ID, the app’s package name (such as `com.android.email`), the description, and the encryption key. For example, the description for an SDO associated with a certain thread might be the subject of that thread. In the database (whose schema is included in Figure 5.1), the tuple $\langle \text{app package name, SDO ID} \rangle$ is a phone-wide unique identifier. Although not implemented in our current prototype, the database can use the app user id to restrict access to keys only to the apps that created them. Finally, to enable auditing services such as Keypad [60], CleanOS notifies the cloud asynchronously whenever it evicts an SDO (message `sdoEvicted`). The

notification is needed because, unlike Keypad, CleanOS does not forcibly evict keys at an exact time after they were fetched; rather, it does so when convenient, depending on load and networking conditions (see §5.3.5 and §5.4).

Default SDOs. As mentioned in §5.2 (Goal 3), we aim not to rely on app modifications to gain tangible benefit from CleanOS. To this end, we modified the Android framework to register a set of default SDOs and use simple heuristics to identify and classify Java objects coarsely on behalf of the apps. For example, our current prototype creates several default SDOs by plugging into various core classes in the SDK: a *User Input* SDO for all input a user types into the keypad (class `InputConnection`), a *Password* SDO for any Java objects that capture input a user types into a password field (based on attributes of class `TextView`), a coarse *SSL* SDO for all objects read from incoming SSL connections (class `SSLSocket`), and SDOs for input from particularly sensitive sensors, such as the camera. Some of these heuristics (e.g., SSL) were inspired by prior work on automatic identification of sensitive data [38]. Although default SDOs are coarse and may potentially include many non-sensitive objects (particularly SSL), we believe that they offer comprehensive identification of most sensitive data in unmodified apps. For example, all the sensitive data we analyzed in §5.1 would be capturable by a default SDO. For apps willing to adapt, CleanOS allows the overriding of default assignments of objects to SDOs.

Eviction Granularities and Buckets. Thus far, eviction granularities have been determined by SDOs, which is problematic for two reasons. First, it forces app writers to consider granularities and taint propagation when they design their SDOs. Second, our default SDOs, such as SSL, are coarse. In our view, CleanOS should offer eviction benefits

even when an app dumps all of its sensitive objects into one big SDO, e.g., “*sensitive*.”

To support fine-grained eviction with coarse-grained SDOs, we introduce *buckets*. Specifically, an SDO is “split” into several disjoint buckets, which are evicted independently. Java objects added to the SDO – either by the app or by our framework – are placed in random buckets. Eviction occurs at the bucket level: when a bucket has been idle for a period, all objects in it will be evicted using a bucket key, which is derived from the SDO’s key using a key derivation function [83]. For example, in an unmodified email app, we would place all emails into the *SSL* SDO. With buckets, different emails would be placed into different buckets of the *SSL* SDO and might therefore be evicted independently. Also, with bucketing, we cache bucket keys instead of SDO keys on the device.

5.3.3 The Evict-Idle Garbage Collector

A simple but important innovation in CleanOS is the evict-idle GC (eiGC). At its core (and independent of CleanOS), eiGC implements for a managed language what swapping implements for OSes: it monitors when objects are being accessed during bytecode interpretation and evicts them when they have not been used for a while. We believe that the eiGC concept has applications beyond CleanOS, such as limiting the amount of memory used by Java applications on memory-constrained devices at a finer grain than OS-level paging would be able to sustain. In the context of CleanOS, however, eiGC evicts Java objects in idle SDO buckets.

Using the GC to evict sensitive data is not the only design worth considering when building a “clean” OS. We contemplated modifying the kernel’s paging mechanism to

swap idle pages to a Keypad-like encrypted file system [60], which at its core achieves for files a similar eviction function to the one we achieve for RAM. We chose the GC approach for two reasons. First, evicting Java objects provides finer-grained control over sensitive-data lifetime than full-page eviction. Second, by evicting at the JVM level we can leverage TaintDroid, the only taint tracking system for Android. Tracking sensitive data is vital for constructing the SDO abstraction, which in turn is the base for building powerful add-on services, such as auditing. However, our decision has a downside: coverage. By evicting Java objects, we may miss data intentionally maintained by native libraries. We discuss this limitation further in §5.7.

5.3.4 SDO Extension to Stable Storage

Like RAM, stable storage requires sanitization. At first glance, systems such as Keypad [60] could be directly leveraged to evict unused files in CleanOS. Unfortunately, we found that eviction at file granularity is unsuitable for Android, where apps typically rely on a database layer to manage their data. For example, 11 of the 14 apps in Figure ??(b) store their data in SQLite, which maps entire databases as single files in the FS. As a result, if the DB file were exposed, then *all* of its items would be exposed, including long-unaccessed emails and documents.

CleanOS tailors storage eviction specifically for Android by extending the in-RAM SDO abstraction to include files *and* individual database items. For this, we use two mechanisms. First, we propagate SDO taints to files and database items. Unfortunately, TaintDroid supports only the former, not the latter, an important vulnerability we discuss in

§5.4. We fixed this in CleanOS by modifying the SQLite DB. Specifically, we automatically alter the schema of any table to include for each data column, C , a new column, $Taint_C$, which stores the taint for each item in that column (SDO ID and bucket ID). Second, before storing a tainted data object in a DB, we first *evict* that object, i.e., encrypt it with its eviction key. When the database needs the object, it must decrypt it.

5.3.5 Disconnected Operation

While we assume that disconnection is the exceptional case, we present techniques to deal with two types of disconnection: (1) short-term disconnection, such as temporary connectivity glitches, and (2) long-term, predictable disconnection, such as a disconnection during a flight. To address short-term disconnection, we can extend eviction of already available SDOs by a bounded amount of time (e.g., tens of minutes). This allows an app to continue executing normally while temporarily disconnected until it reaches an evicted object. For example, a user might be able to load recently accessed emails, but not older ones.

To address long-term disconnection, such as during air travel, we hoard SDO keys before entering into disconnection mode. For example, our prototype implements Dalvik support for hoarding SDO keys upon receipt of a signal. We plan to wrap this functionality into a privileged app that provides users with a “Prepare for Disconnection” button, which they can press before boarding a flight. To prevent a thief from using this button to retrieve all SDO keys, the cloud would require the user to enter a password. While we generally shun user-configured passwords in CleanOS, we believe that long-term disconnection is

a sufficiently rare case to warrant enforcement of particularly strong password rules with limited impact on usability [95]. In contrast, imposing such rules on frequent unlock operations would be impractical.

5.3.6 Deployment Models

CleanOS presents multiple deployment opportunities. First, security-conscious apps can use their own, dedicated clouds to host keys and provide add-on services, such as auditing. In such cases, we expect that the mobile side of apps would define meaningful SDOs. Second, users who are particularly concerned with apps that have not yet integrated with CleanOS might use a CleanOS cloud offered by a third party or that they host themselves. For example, our prototype hosts all keys for all apps on a Google App Engine service that we implemented.

5.4 Prototype Implementation

We built a CleanOS prototype by modifying Android 2.3.4 and TaintDroid in significant ways (see Figure 5.1). To date, our prototype fully implements eviction of in-memory SDOs and propagates taints to SQLite, but it does not yet encrypt sensitive items in SQLite. Doing so will require changing the native part of the SQLite library – a single, massive, over-100K-LoC file – the major deterrent we encountered thus far. We next describe modifications we made to components of particular interest.

TaintDroid with Millions of Taints. Most dynamic taint-tracking systems, including TaintDroid, support limited numbers of taints, which would prevent CleanOS from

scaling to many SDOs. For example, TaintDroid supports only 32 taints by representing them as 32-bit shadow tags, where each taint corresponds to one tag bit. This limitation allows propagation of multiple taints on one object for tracking completeness and security against malicious applications. For CleanOS, which trusts applications, we modified propagation to allow many taints.

We rely on a simple observation, which we validate experimentally: in practice, when multi-tainting occurs, we can usually define a strict, natural ranking for taints in terms of their sensitivity. As intuitive examples, a *Password* SDO should be more sensitive than a generic *User Input* SDO, and a KeePass secret’s SDO should be more sensitive than its description SDO. In these cases, “losing” the less sensitive taint would be admissible, because it does not weaken the user’s perception of the gravity of an object’s exposure. Using a 24-hour real-usage trace for the Email app (see §5.6.1), we confirmed that 98.8% of the tainted objects were either assigned a single taint during their lifetimes or received multiple taints whose sensitivity could be strictly ordered using a simple, static, three-level ranking system: HIGH, MEDIUM, and LOW. The remaining 1.2% of the objects received multiple taints of undecidable ordering within this ranking system (i.e., equal sensitivity levels). Similar traces for Facebook and Mint indicated even fewer undecidable cases ($< 0.01\%$).

Based on this observation, we introduce the concept of *sensitivity level* for taints and use it to propagate a single taint per object. Apps specify a sensitivity level for each SDO upon its creation. If an object were added to two SDOs during taint propagation, CleanOS retains the one with the higher sensitivity level. For equal sensitivities (the rare case), CleanOS retains the most recent taint. Figure 5.3 illustrates the revised structure for



Figure 5.3: *CleanOS Taint Tag Structure*. We impose a structure on TaintDroid taints to support arbitrary numbers of taints.

the taint tag, in which we pack together the sensitivity level, SDO ID, and bucket ID into 32 bits while supporting up to 2^{25} SDOs. In our experience, assigning sensitivity levels to SDOs is natural, as demonstrated in §5.5. The idea of propagating a single taint was used before in hardware-based taint tracking systems for improved performance [27].

Eviction-Aware Interpretation in Dalvik. We reserved the most significant bit in the taint tag to denote the eviction state of a field. We modified the Dex bytecode instructions that access object instance fields and array members. This includes instructions such as `OP_AGET`, `OP_IGET`, `OP_SGET` (used to retrieve array members, instance fields, and static fields, respectively). Our new instruction implementations first test the value of the eviction bit in the field’s taint tag. When the bit is set, we request the aforementioned K_{SDO} and decrypt the value before allowing the instruction to proceed. If a key is not available, execution is suspended.

The Evict-Idle Garbage Collector. While eiGC walks the reachable objects, we inspect the taint tag for each object field and retrieve its idle time. If it exceeds the configured threshold, then eiGC retrieves the key associated with the tag and encrypts the value. Only fields that represent actual data are evicted (primitives and arrays of primitives); fields implemented as pointers are not evicted, as a pointer is not in and of itself sensitive.

To evict data, we use AES in counter mode to generate a keystream, which we use

as input to an XOR operation with each byte of the data to be evicted. The size of the keystream depends on the data's type. For primitives, it is either 4 bytes (for char, int, float, etc.) or 8 bytes (for double or long). For arrays, many bytes may be necessary. We use the bucket key to generate an appropriately sized keystream. For primitives, we replace the data with a pointer to a structure containing metadata necessary for decryption (e.g., initialization vectors) and the resulting ciphertext. For arrays, we evict the contents in place and store the necessary metadata inside the ArrayObject.

Running the eiGC continuously would prevent the CPU from turning off when the mobile device is idle, thereby wasting energy. Fortunately, eiGC needs to run only while sensitive objects are left unevicted. Hence, in our prototype, eiGC stops executing as soon as it has evicted all data, which should occur shortly after the app goes idle. The eiGC resumes execution once the app faults on an evicted object or assigns a new taint to an object. Hence, eiGC runs only while the app also runs.

Optimizations: Bulk Eviction and Prefetching. Performance and energy are major concerns with CleanOS, for two reasons. First, garbage collection is expensive; hence performing it frequently hurts app performance and energy (e.g., the eiGC's full-heap scans block interpretation for 1-2s). Second, our reliance on the network to fetch decryption keys causes app delays and dissipates energy.

To address the first problem, we developed *bulk eviction*, in which the eiGC evicts sensitive Java objects *all at once*, soon after the app itself becomes idle. More specifically, while the app is executing, we evict nothing and perform no GC; once the app has remained idle for a predefined time (e.g., one minute), the eiGC performs a full-heap

scan-through and evicts all cleartext tainted objects. This technique reduces the number of heavyweight GCs to just one per app execution session, thereby minimizing the eiGC's impact on performance and energy.

To address the second problem, we developed *bulk key prefetch*, which prefetches all keys that were accessed during the last eviction period upon the app's first miss on a key. For example, if a user opened his inbox subject list and read two emails during a previous interaction session with his email app, then the next time the user brings the app into the foreground, CleanOS will fetch the decryption keys for the subjects and the two emails' contents – all in one network request. If the user views only his subject list but reads no emails in a previous session, then the next time around, CleanOS will fetch only subject keys again, not any email content keys. This technique improves app launches and the latency of repeated operations, such as re-reading an email. It can be extended to prefetch keys used in the last N sessions.

Although these optimizations may improve performance and energy, they may also increase sensitive-data exposure. For example, prefetching previously-used keys may expose some sensitive data needlessly. We quantify this performance/exposure tradeoff in §5.6.2.

Multi-Level Secure Memory Deallocation. Android goes to great lengths to keep an application running in the background so it can re-launch quickly. This can cause an accumulation of sensitive data in areas of memory that are no longer in use but have not been returned to the kernel. The object heap in Dalvik is implemented using `dlmalloc` mspaces and relies on the implementation of `free()` in `dlmalloc` to return memory to the

mspace. To implement secure deallocation, we changed both `free()` and an Android-specific modification to `dlmalloc` that merges chunks of adjacent free memory. These functions now overwrite the space being released with a fixed pattern. We also modified Dalvik to overwrite interpreted stack frames on method exit, scrubbing them of sensitive data. Finally, when assigning default taints to Java objects, we made explicit efforts to taint objects as soon as they enter Java space from native libraries.

Addressing a TaintDroid Vulnerability. When implementing CleanOS, we uncovered a surprising implication of a known limitation in TaintDroid. Specifically, TaintDroid does not track changes in native libraries, which, as acknowledged by its authors, may allow a *malicious* library to leak tainted data without triggering an audit log. To address this problem, TaintDroid prevents untrusted apps from loading any native libraries other than system libraries (e.g., SQLite and WebKit), which are included in Android itself and are therefore *trusted*. This measure has thus far been thought sufficient.

Nevertheless, we discovered that even *trusted* system libraries can be exploited by a malicious app to expose tainted data with no alarms. For example, because SQLite is written in native code, a malicious app could wash taints off a tracked data item simply by storing it into the database and reading it back. More generally, any stateful libraries that provide the ability to put and later retrieve data are vulnerable to attacks. Since disabling system libraries is impractical (e.g., 12/14 apps in §5.1 depend on SQLite), we instead suggest identifying and modifying all stateful system libraries to propagate taints.

To date, we modified two such libraries: SQLite and WebKit. For SQLite, we implemented taint propagation by persisting taints along with the data (see §5.3.4). For WebKit,

we disabled caching of rendered Web pages. While important for security, we leave identifying and fixing other libraries for future work and for now suggest notifying the cloud about a potential leak if sensitive data were handed over to an unchecked native library. We suggest that TaintDroid proceed similarly. We discuss the coverage limitation further in §5.7.

5.5 Applications

We ported three of our “dirtiest” apps from §5.1 onto CleanOS and built a proof-of-concept, add-on service.

5.5.1 Extending Apps with SDOs

Although unmodified apps can benefit from the coarse default SDOs that CleanOS offers, they can also define their own SDOs for fine-grained control of sensitive data. To demonstrate how apps can be “ported” to our API, we modified two open-source apps – Email and KeePass – to define fine-grained SDOs. Changes for both apps were trivial. For Email, we added these seven lines of code:

```
SDO subjectSDO = new SDO("Subject", SDO.LOW);
subjectSDO.add(mSubject);

SDO bodySDO = new SDO("Content_of_" + mSubject, SDO.MED);
bodySDO.add(mTextContent);
bodySDO.add(mHtmlContent);
bodySDO.add(mTextReply);
```

```
bodySDO.add(mHtmlReply);
```

We added each email’s subject to a global, low-sensitivity SDO and created a medium-sensitivity content SDO for its body, using the subject itself as the description. Passwords, already embedded in an SDO by our default heuristics, needed no changes.

For KeePass, changes were similarly trivial (7 lines):

```
SDO masterSDO = new SDO("Master_key", SDO.MED);  
SDO entrySDO = new SDO("Entry", SDO.HIGH);  
masterSDO.add(mPassword); // In SetPassword.java  
masterSDO.add(masterKey); // In PwDatabase.java  
entrySDO.add(password); // In PwEntryV3.java  
entrySDO.add(pass); // In EntryEditActivity.java  
entrySDO.add(conf); // In EntryEditActivity.java
```

5.5.2 Add-on Cloud Services

CleanOS evicts sensitive data to the cloud to prevent unmediated accesses by device thieves. However, by itself, CleanOS cannot guarantee data security. For example, a thief could interact with the apps in an unlocked device or force all SDOs to decrypt. Therefore, CleanOS provides device-side mechanisms necessary for clouds to build clean-semantic security add-ons, such as assured remote wipeout or data exposure auditing. Such services already exist today (e.g., Apple’s iCloud and Gmail’s two-step verification), but we maintain that their semantics are unclear given the state of today’s devices. We

device	message	time
cd5493c1befeb9075442862afa046182	fetchKey(9.408 - com.android.email - password)	2012-04-28 17:26:50.590000
cd5493c1befeb9075442862afa046182	registerSDO(com.android.email - Invitation to develop "Clean OS")	2012-04-28 17:27:01.140000
cd5493c1befeb9075442862afa046182	fetchKey(30.709 - com.android.keepass - Entry)	2012-04-28 17:27:48.500000

Figure 5.4: *Screenshot of Audit Service Log in App Engine.*

next describe an add-on service we trivially built on CleanOS.

Prototype Auditing Service. Inspired by Keypad [60], we implemented an auditing service on CleanOS. Its goal is to provide users with audit logs of what was on the device at the time of theft and what has been accessed since. The auditing service integrates with the CleanOS service and both are hosted on App Engine. When a device registers an SDO or requests a decryption key, the cloud logs that operation with the app name, SDO, and current time. In this way, the user can learn from the audit log exactly what data was leaked. For instance, Figure 5.4 shows a sample audit log that contains entries for SDO registration and key fetching. Were these operations to occur after the device was stolen, the user will know that the email password and KeePass entry may have been leaked.

Crucial to any auditing system is precision. In the audit log, data in different buckets of the same SDO are indistinguishable. Thus, accessing the data in one bucket may cause false alarms for evicted buckets of the same SDO. Using a finer SDO granularity helps reduce false positives. We evaluate audit precision in §5.6.1.

Further Examples. A cloud could build many other useful services on CleanOS. For example, the cloud could: allow its mobile users to revoke data access from their missing

devices, disable access to sensitive data while the phone is outside the corporate network, and perform theft detection based on access patterns. A variety of entities would find such services useful to host. For example, a company might integrate with CleanOS on the device for all corporate apps (e.g., corporate email, customer database), to access its auditing, revocation, and geography-constrained services. Similarly, Gmail could integrate with CleanOS to prevent email exposure after authentication-token revocation.

5.6 Evaluation

We next quantify CleanOS’ security, performance, and energy characteristics. Our goal is to show that CleanOS significantly reduces sensitive data exposure while providing reasonable performance and energy consumption, even over cellular networks. We conducted all experiments on rooted Samsung Nexus S phones running CleanOS on Android 2.3.4 and TaintDroid 2.3.

5.6.1 Data Exposure Evaluation

To evaluate the data exposure benefits of CleanOS, we pose three questions: How much does eviction limit exposure of sensitive data? How much do default SDO heuristics limit exposure? How effective is the auditing service? To answer these questions, we recorded a 24-hour trace of one of the authors’ phone running CleanOS as it was used to interact with regular apps, including Email, Facebook, and Mint. For Email, we experimented with both the unmodified app and our modified version of it, which we call CleanEmail (see §5.5.1). The Email app was configured with the author’s personal account, which receives

	Password	Content	Metadata
Email without CleanOS	100%	95.5%	99.0%
Email with default SDOs	6.5%	5.9%	5.9%
CleanEmail (fine SDOs)	6.5%	0.3%	1.6%

Table 5.5: *Sensitive data exposure period.* Numbers are the fraction of time in which sensitive data was exposed.

about ten new mails daily, and with the default 15-minute refresh period. Facebook and Mint had widgets enabled, which made them continuous services.

Sensitive Data Exposure Period. We measured the exposure period for three types of tainted data (password, content, and metadata) in the Email app. Table 5.5 shows the fraction of time that each type of tainted data was exposed in RAM. Without CleanOS, the password was maintained in RAM all the time, and the content and metadata were exposed over 95% of the time. CleanOS reduced password exposure to 6.5%. For email content, the unmodified Email app with default SDOs reduced exposure time from 95.5% to 5.9%, and modifying the app to support fine-grained SDOs further reduced it to 0.3%. Similar observations held for metadata. To be clear, these results depend on workloads. From another, much more intensive email workload – that registered for many mailing lists and Twitter feeds – we obtained a result of 7.3% and 12.7% for content and metadata, respectively. Overall, results demonstrate a significant reduction in exposure times for tainted data. Moreover, they show that our default heuristics protect sensitive data reasonably well.

Sensitive Data Lifetime. As SDO lifetime is critical to system security, we must also examine the maximum period that a tainted object could be retained in RAM. Table 5.6

App	Without CLEANOS	With CLEANOS		
		1 bucket	32 buckets	1024 buckets
Email (password)	22.5h	1h 28m	1h 19m	1h 11m
Email (contents)	20.9h	3 min	1 min	1 min
Email (Metadata)	20.9h	18 min	6 min	6 min
Facebook	24h	3h 54m	3h 51m	3h 29m
Mint	24h	1h 10m	1h 2m	55 min

Table 5.6: *Sensitive data lifetime*. Numbers are the maximum sensitive data retention period.

shows the retention time for the longest-lived tainted object in three applications, where we break down email into three types. Without CleanOS, all observed applications retained certain tainted objects for more than 20 hours. With CleanOS, the maximum SDO lifetime was dramatically reduced. For instance, the Email app kept some metadata objects for as long as 20.9 hours, which CleanOS reduced to only 6 minutes when using 1024 buckets. For Facebook and Mint, the impact of bucketing on sensitive data lifetime was more limited because these apps tend to use most objects in an SDO at the same time. Overall, these results indicated that the mobile device was significantly cleaner with CleanOS.

Audit Precision. We next evaluated the effectiveness of the auditing service we built on CleanOS (see §5.5.2). We compared audit precision across four levels of SDO granularity in Email: (1) mono-SDO, where we marked data as only “sensitive” or “non-sensitive,” (2) default SDOs, where we used default heuristics, (3) coarse SDOs, where the application defined one content SDO and one metadata SDO for all emails, and (4) fine SDOs, where each email had its own content and metadata SDOs. We define *audit precision* as the average probability over time that the tainted data is actually exposed on the device, given

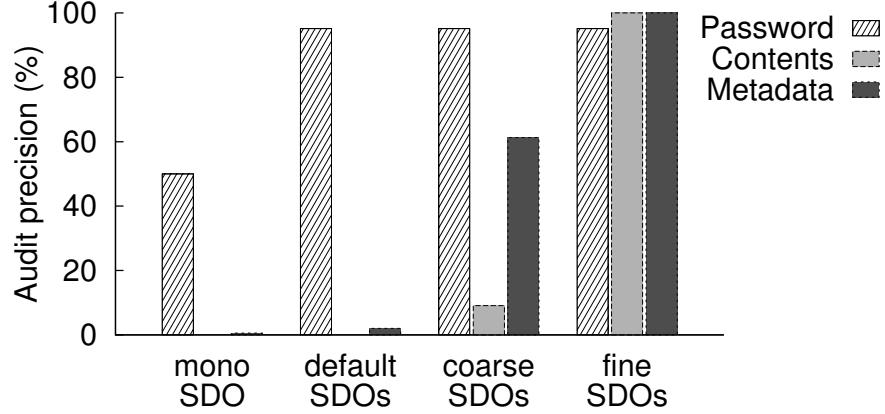


Figure 5.5: *Audit precision*. Each bar shows the average probability over time that tainted data was actually exposed, given that the audit log shows its SDO as exposed.

that the audit log shows its SDO has not been evicted.

Figure 5.5 shows audit precision for the Email app’s password, content, and metadata. Password auditing was 50.0% precise with mono-SDO but increased to 95.1% with default SDOs. The content and metadata, however, had poor precision (<3%) without application support: CleanOS could not differentiate data coming from the Internet and hence added every incoming object to the *SSL* SDO. With coarse, application-specific SDOs, audit precision for email content and metadata was 9.1% and 61.3%, respectively. When fine application-specific SDOs were available, audit precision reached 100%. Thus, our default SDOs were effective in auditing password exposure, but application adaptation was needed to provide precise auditing for other types of sensitive data.

5.6.2 Performance Evaluation

We next evaluate the performance impact of CleanOS under different workloads and networking conditions. Here, we aim to: (1) quantify raw performance overheads, (2)

	Android 2.3.4	TaintDroid 2.3	CleanOS			
			not evicted	evicted, cached	evicted, Wi-Fi	evicted, 3G
Untainted Primitive	0.00021	0.00022	0.00026	-	-	-
Tainted Primitive	-	0.00023	0.00056	1.24	22.844	336.07
Untainted Array	0.00027	0.00029	0.00035	-	-	-
Tainted Array (S)	-	0.00030	0.00075	1.4	21.652	308.71
Tainted Array (M)	-	0.00030	0.00075	1.331	21.702	316.79
Tainted Array (L)	-	0.00030	0.00075	2.355	22.365	317.97

Figure 5.6: *Micro-operation Performance (milliseconds)*. CleanOS Java object field access times compared with Android, TaintDroid. Times for non-sensitive and sensitive fields for various eviction states. Averages over 1,000 accesses.

demonstrate that CleanOS is practical over Wi-Fi for popular apps, and (3) show how our optimizations make CleanOS practical even over slow, cellular networks. In our experience, obtaining reliable and repeatable results from the cellular network is tremendously difficult; hence, our results used emulated Wi-Fi and 3G networks with RTTs configured at 20ms and 300ms, respectively. Because our transmission units were tiny (keys were 16-byte long), we did not enforce bandwidth restrictions.

Micro-operation Performance Overheads. To evaluate raw performance overheads, we measured Java object field-access times for Android, TaintDroid, and CleanOS. Figure 5.6 compares them for four field types: primitives (int), small arrays (16 bytes), medium arrays (4KB), and large arrays (16KB). For CleanOS, we show access times both for non-sensitive fields (the vast majority) and sensitive fields under various eviction states. CleanOS’ access overhead for non-sensitive fields was small compared with TaintDroid (16%), which itself was close to raw Android (6% overhead for TaintDroid). The overhead for sensitive field access increased to 141% over TaintDroid: CleanOS performed

Application	Action	Android 2.3.4	TaintDroid 2.3	CleanOS			Optimized CleanOS
				not evicted	evicted, Wi-Fi	evicted, 3G	
Email	Launch	197	202	241	312	919	589
	Read Message	212	254	387	501	1165	379
CleanEmail	Launch	-	-	291	315	902	598
	Read Message	-	-	452	526	1472	421
KeePass	Launch	173	192	217	221	527	672
	Read Entry	125	150	146	155	479	135
Browser	Launch	130	151	160	144	222	138
	Load Page (iana)	Wi-Fi	488	483	658	605	-
		3G	2067	2114	2125	-	2136
	Load Page (GNews)	Wi-Fi	1072	1043	1270	1160	-
		3G	1717	2475	2475	-	3536
	Load Page (CNN)	Wi-Fi	1065	1136	1394	1446	-
		3G	4570	4709	4325	-	4619

* Actions were performed before.

Figure 5.7: *Application Performance*. Performance of various popular app activities under Android, TaintDroid, and CleanOS for various eviction states and configurations. Results are averages over 40 runs, in milliseconds.

last-time-of-use bookkeeping *on every Dalvik field access instruction* (e.g., OP_AGET, OP_IGET) that involved a tainted field. Further, when evicted, CleanOS access overhead spiked dramatically, especially when the evicted field’s key was not cached on the device but was fetched over Wi-Fi or 3G. Moreover, unlike in Android and TaintDroid, access times for evicted arrays in CleanOS depended on the array’s size because decryption times increase with data size. For example, the “evicted, cached” column shows that decrypting a tainted array grew by 68% when the array’s size increased from 16B to 16KB. Fortunately, in practice, sensitive fields are extremely rare compared with non-sensitive fields. For example, our email trace showed an average of 102,907 fields at any time, of which merely 1,889 were tainted (or 1.83%). Hence, CleanOS should acceptably affect real app performance, as shown next.

Application Performance. Figure 5.6.2 shows the time to launch several popular apps

(i.e., bring them into the foreground) and perform typical actions, such as opening an email, viewing a KeePass entry, or loading a Web page. We chose three Web pages: a simple one (<https://iana.org/domains/example>) and two popular and more complex ones (<https://news.google.com> and <https://cnn.com>). For CleanOS, results labeled “not evicted” correspond to cases where all accessed objects were decrypted, while results labeled “evicted” correspond to cases where objects were all evicted.

In the “not evicted” case, interaction with the apps incurred a limited performance penalty compared with both TaintDroid and Android. For example, 8/13 operations incurred less than 100ms penalties over TaintDroid, and 7/13 did so over Android. Such penalties will likely go unnoticed by users, who are known to perceive delays coarsely [100]. Hence, when users interact with a recently used app, they should not feel CleanOS’ presence.

When users interact with a cold app (“evicted” columns for unoptimized CleanOS), however, performance degraded but remained usable for Wi-Fi networks. Our cheapest app is the browser, for which CleanOS incurred 8-23% overheads over Android for all operations. The reason is two-fold: (1) the browser deals with little sensitive data, and (2) during page loads, the browser fetches large amounts of data over Wi-Fi, which dwarf CleanOS’ key traffic delays. The most expensive app for CleanOS is CleanEmail, which incurred a larger penalty than Email for “evicted” launches due to more granular tainting. For example, while Email needed to fetch 2 keys to load an email, CleanEmail needed to fetch 3 keys.

Over 3G, CleanOS penalties after eviction became significant. While some operations remained within reasonable bounds (e.g., launching the browser and loading iana.org or

cnn.com), many operations incurred overheads in excess of 100%. For example, loading an email onto the screen jumped from 197ms to 1.1s for Email and 1.4s for CleanEmail. Such delays likely affect usability.

Effect of Optimizations on Application Performance. Column “Optimized CleanOS” in Figure 5.6.2 shows the elapsed time of repeat operations under our bulk prefetching optimization (see §5.4). All timed operations were invoked in the previous application session; therefore, all of their relevant keys were prefetched together as part of one bulk request during the timed session. The results show dramatic improvements in performance for both launching and interacting with the apps. For example, CleanEmail – our most expensive application – launched in 589ms over 3G compared with 919ms on unoptimized CleanOS (35.9% improvement) and loaded a previously read email in 420ms compared with 1.4s (71% improvement). In general, this optimization lets an app re-launch incur little more than one RTT over non-evicted CleanOS, while subsequent repeat operations incur no RTT. Naturally, our optimization will not benefit non-repeat operations, such as loading a brand new or long-unread email. However, one type of operation that will always benefit is app launch, a latency-sensitive operation on mobiles.

Despite their performance benefits, our optimizations may increase data exposure. When applying these optimizations to the workloads in §5.6.1, we obtained limited, but non-trivial, exposure impact. The period for each type of tainted data increased by up to 0.9 percentage points for the workload in Table 5.5, and by up to 23.2 percentage points for our intensive Email workload. Prefetching keys from multiple sessions would cause further exposure. Hence, CleanOS should best apply this optimization only in specific

cases (e.g., over 3G).

Overhead Estimation for SDO Stable Storage Extension. Thus far, our results show CleanOS’ overheads for eviction of *in-RAM SDOs*. While we have not fully implemented the SDO extension to stable storage, we now offer rough estimates for the extra overheads to expect from such an extension. We expect the major sources of overhead to be: (1) the key fetches required to access encrypted database items, and (2) the extra encryption/decryption that occurs when accessing these items. To account for (1), we ran experiments with our test applications that instruct CleanOS to fetch the appropriate decryption keys for any tainted database items being accessed. To account for (2), we added an extra 20% overhead per query, a number reported by CryptDB [111], which also does per-item encryption.

With this methodology, we estimate that extending SDOs to SQLite would result in additional overheads ranging between 0-65% on 3G over CleanOS with in-RAM SDOs. We predict that these operations will suffer the most: KeePass Launch (869ms, or 64.9% additional overhead), CleanEmail Read (1887ms, or 28.2% additional overhead), and Browser Load (2542ms, 4086ms, and 4573ms for `iana.org`, `news.google.com`, and `cnn.com`, respectively, or 15-19% additional overhead). Most of these overheads (82-99% across all apps) are due to extra RTTs incurred by necessary key fetches, which are optimizable via batch prefetching. Thus, overall, we believe that our system will be practical from a performance perspective even when implemented in full.

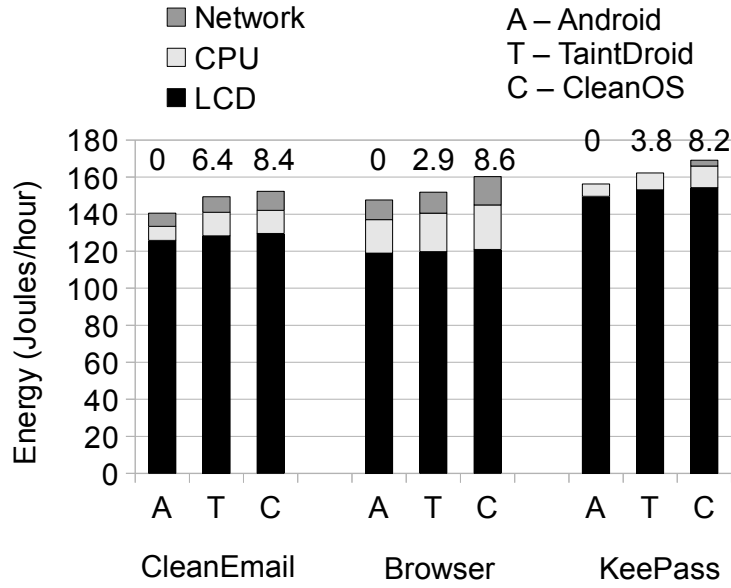


Figure 5.8: *Energy Consumption*. Hourly energy consumption attributed by PowerTutor to the three apps when running a long-term synthetic workload for at least 3 hours over Wi-Fi. Numbers on top of each bar show energy overhead over default Android in percent.

5.6.3 Energy and Network Evaluation

CleanOS’ encryption, network traffic, and extra GCs raise concerns about its impact on energy consumption. To evaluate this impact, we ran coarse-grained experiments that drove a simple, long-term workload against each app (CleanEmail, Browser, and KeePass) using MonkeyRunner [65] and measured consumption using the PowerTutor online power monitor [152]. The workload repeatedly launched an app, performed a set of typical tasks (such as reading emails, accessing entries in KeePass, and visiting Web pages in the browser), sent the app into the background, and then slept for 15 minutes. Each app interaction lasted for 36-46s, after which we promptly turned off the LCD. We ran the workload continuously for at least 3 hours and plotted per-app power consumption as reported by PowerTutor.

Figure 5.6.3 shows energy consumption for Android, TaintDroid, and CleanOS over a *real* home Wi-Fi network. For each app, we show the energy consumed by the LCD, CPU, and Wi-Fi. Results show that CleanOS' total energy overheads over Wi-Fi were small compared with both Android and TaintDroid: 8.2-8.4% over Android (see labels above bars) and 1.9-5.5% over TaintDroid. Drilling down on resource overheads, we observe that CleanOS increased energy consumption of both the network (44-45%) and the CPU (32-74%), but those overheads were dwarfed by the LCD energy draw. In general, our overheads were smallest for the browser, which itself consumed relatively more CPU and network energy, and largest for KeePass, a lightweight application that performed little computation and had no network traffic.

Over 3G, energy overheads due to network traffic will likely increase. Our experience shows that experimenting with 3G networks leads to very unstable and unrepeatable results; hence, for these networks, we rely on an analytic evaluation grounded in a study of CleanOS' network traffic. Figure 5.9 compares CleanOS' traffic patterns to those of the three apps using one-hour traces from our energy experiments. It shows that CleanOS' network consumption depends on the application's own network profile. For networked apps, such as email and browser, CleanOS' traffic closely follows the app's own traffic distribution over time. For example, for email, of the 24 minutes during which CleanOS issued some traffic, only 9 of those had no accompanying app traffic; for the browser, only 1 out of 5 one-minute periods did so. From an energy perspective, this means that CleanOS usually piggybacks on the app's own use of the network and only rarely needs to hold the interface up for its own purposes. On the other hand, for local-only apps, such as KeePass, CleanOS uses the network mostly for its own purpose; but even in such cases,

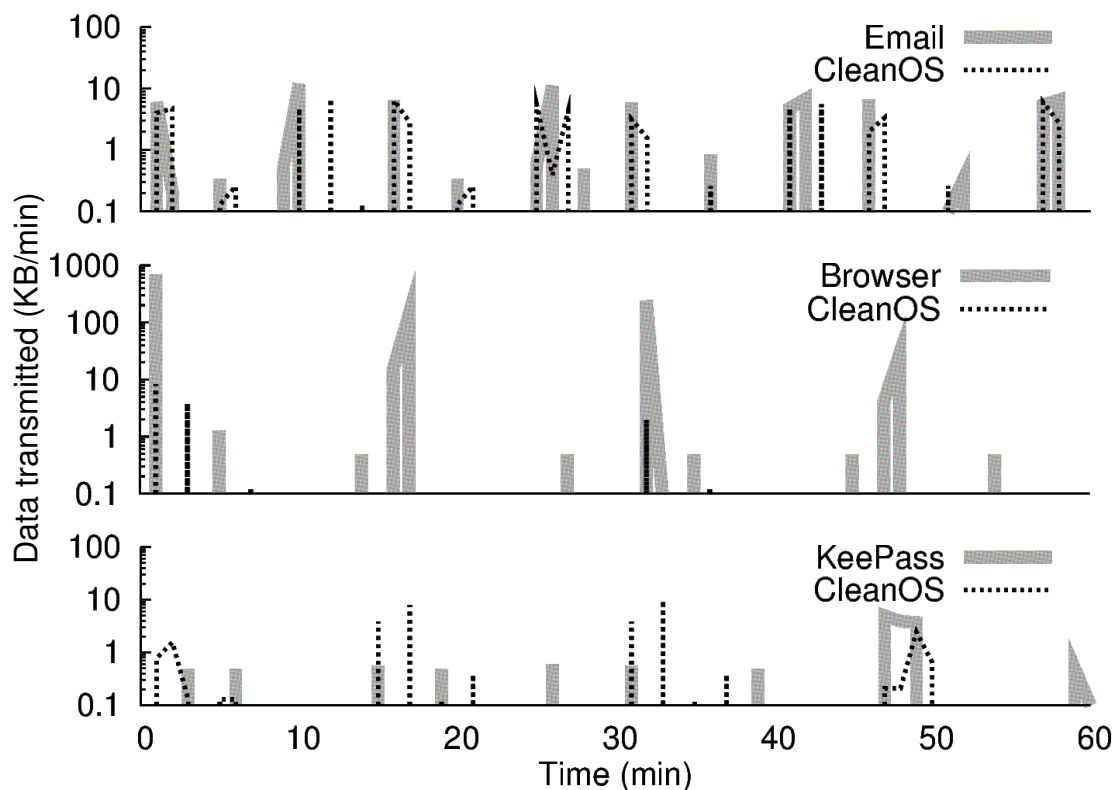


Figure 5.9: *Network Traffic Patterns of Apps vs. CleanOS*. CleanOS traffic vs. app traffic for a one-hour trace. The Y axis is in log scale. In our cases, the phone has background traffic, which is included in both app and CleanOS lines.

however, its traffic will be rare, brief, and small ($\leq 10KB/min$). Thus, we expect CleanOS to be practical from an energy perspective.

5.7 Security Discussion and Limitations

We now discuss CleanOS' security implications and limitations. There are two types of data that an attacker might seek: unevicted data and evicted data. CleanOS does not protect unevicted data on a stolen device; instead, it seeks to minimize the amount of such data. An audit-enabled cloud service can provide users with a robust audit trail of data exposed at the time of loss and data retrieved since. For evicted data, clouds can

do much more. For example, after theft has been detected, they can revoke the device's access to still evicted data. They can also monitor accesses to keys to detect anomalous behavior.

A thief might also try to retrieve keys for all evicted SDOs before the cloud disables them. Such aggressive attackers could be identified via anomalous access-pattern detection. To evade detection, the attacker could retrieve SDO keys only for objects of interest, such as emails with tempting subjects. While some attackers may be unwilling to do so for fear of revealing their identities, the cloud can provide an audit log of such accesses.

Attackers might also attempt to break the disconnection password to hoard keys for apps of interest without raising suspicion. CleanOS could enforce sufficient entropy to make the disconnection password, which is extremely rarely used, much stronger than a regular password (which a user must type every time he unlocks his device). However, even if the password were broken, the cloud could provide evidence of the attacker's behavior.

Adversaries may perform network attacks to sniff or disrupt CleanOS device-cloud traffic. To prevent sniffing of keys from network traffic, we encrypt connections and authenticate the device to the cloud using a pre-established secret key (akin to the device token in Gmail's two-factor verification) and the cloud using public key cryptography. An attacker could also disrupt CleanOS device-cloud communication to induce CleanOS into an accumulation mode, where it defers eviction until cloud connectivity returns. To defend, CleanOS bounds its eviction delay for temporary disconnections. Moreover, a thief could prevent eviction messages from arriving at the cloud. However, dropping those messages will not affect confidentiality since data eviction will complete as planned, but

it might raise auditing false positives.

One CleanOS limitation is its limited coverage outside the Java realm. To be clear, expunging sensitive data from Java is an important contribution: 9/14 apps in Figure ??(b) would expose some sensitive data permanently in RAM if we did not do so. Moreover, we have incorporated some basic multi-level secure deallocation techniques and have modified two popular native libraries to limit exposure (SQLite and WebKit). However, any data retained in other buffers or caches in the OS or native libraries remains exposed. To limit this exposure, we recommend: (1) incorporating additional OS data scrubbing mechanisms [45], (2) inspecting all remaining system libraries for caches as we do for SQLite and WebKit, and (3) either disabling all third-party libraries (an approach similar to TaintDroid’s [49]) or informing the cloud about any data leakages to uninspected third-party libraries.

5.8 Related Work

CleanOS builds upon prior work that we now describe.

Encrypted File Systems. Encrypted file systems [48] and full-disk encryption [96, 129] are designed to protect data stored on a vulnerable device, but they do not protect data in RAM. Moreover, as discussed in §5.2 (Threat Model) and in prior work [60, 141], these systems can fail in the real world due to human factors (e.g., non-existent or poor passwords) and physical attacks (e.g., key retrieval from RAM via cold-boot attacks [70]). CleanOS recognizes these limitations and promptly removes unused data from the vulnerable device.

Encrypted RAM Systems. Encrypted RAM systems – such as XOM [86], CryptKeeper [107], and encrypted swap [112] – encrypt data while it sits in RAM. CryptKeeper resembles the CleanOS model by encrypting all memory pages except for a small working set, thereby achieving a similar encrypted-unless-in-use effect as CleanOS. However, while the data is encrypted in these systems, the decryption keys themselves are still available in RAM and potentially accessible to memory-harvesting unless extra hardware is deployed. Moreover, if the device were unlocked or the thief found the user’s password, encrypted RAM would have no effect.

ZIA [37, 36] encrypts mobile data in RAM and on disk whenever a device is not near its owner. The user wears a beaconing token at all times, whose presence is detected by the mobile. Like ZIA, CleanOS encrypts data after a period of non-use, but the granularities, method, and usage model are different. For example, we disable unused data at the Java object level as opposed to the device level, evict data to clouds for increased post-theft control, and do not require users to carry (and secure!) tokens.

Mobile Wipe-Out Systems. Varied commercial wipe-out systems exist and help increase users’ post-theft data control. For example, remote wipe-out systems, such as iCloud [12], let the users send “kill” messages to lost devices. Unfortunately, these systems require network connectivity to function correctly. If the thief prevents device connectivity (e.g., by wrapping it into a Faraday cage), the device will not receive the message and therefore not complete its wipeout. Moreover, configuring the device to self-destruct after a number of failed authentication attempts helps prevent access to file system data, but it does not preclude memory harvesting attacks, such as coldboot imaging [70]. Such

attacks are particularly problematic on mobile devices, which hardly ever power off.

Cloud-based Mobile Security Services. The value of the cloud for increased data control is being increasingly recognized. Examples of cloud-based security services include: online data access revocation with two-step verification [67], location-based access control with location-aware encryption [128], and cloud-based authentication with capture-resilient cryptography [89]. Generally, these systems prevent the compromise of data not already exposed on the device, but they do not guarantee security for mobile-resident data. For example, none of these systems takes RAM-resident data into account, and the Google two-step verification does not even consider storage. CleanOS cleanses device RAM and storage in support of such security services.

Keypad. Particularly relevant is Keypad [60], an auditing file system for old-generation mobile devices, such as laptops and USB sticks, that achieves file-level, strong-semantic auditing. CleanOS shares Keypad’s threat model, and our auditing service was inspired by it. However, in addition to its support for in-RAM data auditing, CleanOS also differs from Keypad in its focus on new-generation mobile technologies, such as Android, which have distinct auditing granularity requirements. For example, file-level auditing in Keypad would be ineffective for apps using the SQLite database since they all would be stored within one single file. Instead, CleanOS defines SDOs, an abstraction that encompasses fine-grained objects, database items, and sdcard files.

Secure-Deletion Systems. Secure deletion has been recognized as a key OS primitive. It erases data in memory [29], OS buffers [45], and stable storage [106, 131, 23] once the

data is not needed by the application. CleanOS explicitly assumes the existence and robustness of such systems, but addresses a distinct, important part of the sensitive data exposure problem for the first time: securing data explicitly hoarded by applications for performance or convenience. CleanOS SDOs resemble the self-destructing data abstraction in Vanish [61] in that they “disappear” over time, but the setting is different: Vanish makes Web data disappear after a specified time post-creation, whereas SDOs make mobile data disappear if they are unused for a specified time.

5.9 Summary

This chapter presented CleanOS, a new design for the Android OS that manages sensitive data rigorously and keeps mobile devices clean at any point in time. Unlike Android, which lets sensitive data accumulate in cleartext RAM and on disk, CleanOS eliminates it from the vulnerable device by evicting it to the cloud whenever it is not needed on the device. It provides a clean-semantic foundation for clouds to build add-on services, such as data access revocation after a device has been lost or post-theft data exposure auditing. We implemented CleanOS by instrumenting Android’s Java virtual machine to securely evict sensitive data objects after a specified period of non-use. On top of CleanOS, we built a sample auditing cloud service. Our experiments demonstrate that CleanOS limits data exposure significantly while imposing acceptable performance overheads and offering sound semantics for cloud-based applications.

Chapter 6

Conclusion

This dissertation presented a novel approach to make data storage efficient in the era of cloud computing, by building new storage abstractions and systems that bridge the gap between modern cloud computing techniques and legacy data storage solutions, and in the meantime, simplify development.

We have built four systems to solve four data inefficiencies in cloud computing. The first system, GRANDET, is a unified, economical object store for web applications. It serves as a layer between web applications and cloud storage, significantly reducing storage costs for web applications deployed in the cloud.

The second system, UNIC, is a system that securely deduplicates general computations. It serves as a memoization layer that allows applications running on behalf of mutually distrusting users to memoize and reuse computation results, thereby improving performance.

The third system LAMBDATA, is a serverless computing system optimized by making data intents explicit. It adds a cache layer between cloud functions and cloud storage, enabling a variety of optimizations to improve performance and reduce cost.

The fourth system CLEANOS, is a mobile operating system that limits data exposure with cloud-based idle eviction. It adds to the mobile device a cloud-based data manage-

ment layer that manages sensitive data rigorously and maintains a clean environment at all times.

Bibliography

- [1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. “RACS: a case for cloud storage diversity.” In: *Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10)*. 2010.
- [2] Sandip Agarwala, Divyesh Jadav, and Luis A Bathen. “iCostale: Adaptive Cost Optimization for Storage Clouds.” In: *2011 IEEE International Conference on Cloud Computing (CLOUD '11)*. 2011.
- [3] Eirikur Agustsson and Radu Timofte. “NTIRE 2017 Challenge on Single Image Super-Resolution: Dataset and Study.” In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 2017.
- [4] Airbnb. *BinaryAlert*. <https://github.com/airbnb/binaryalert>.
- [5] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. “Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors.” In: *Proceedings of the USENIX Security Symposium*. 2009.
- [6] Amazon. *Amazon EBS Product Details*. <http://aws.amazon.com/ebs/details>.
- [7] Amazon. *Amazon EBS Volume Types*. <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html>.
- [8] Amazon. *Startups and Amazon Web Services*. <http://aws.amazon.com/start-ups>.
- [9] *An Amazon Web Services C++ Library*. <http://libaws.sourceforge.net>.
- [10] Ross Anderson and Markus Kuhn. “Tamper resistance: A cautionary note.” In: *Proceedings of the USENIX Workshop on Electronics Commerce*. 1996.
- [11] Android Developers Blog. *Avoiding Memory Leaks*. <https://android-developers.blogspot.com/2009/01/avoiding-memory-leaks.html>. 2009.
- [12] Apple iCloud. *Find my iPhone, iPad, and Mac*. <https://www.apple.com/icloud/features/find-my-iphone.html>. 2012.

- [13] W. A. Arbaugh, D. J. Farber, and J. M. Smith. "A Secure and Reliable Bootstrap Architecture." In: *Proceedings of the 1997 IEEE Symposium on Security and Privacy (SP '97)*. 1997.
- [14] William A. Arbaugh, Angelos D. Keromytis, David J. Farber, and Jonathan M. Smith. *Automated Recovery in a Secure Bootstrap Process*. 1998.
- [15] Matt Asay. *Amazon Web Services leads war on cloud price reductions*. <http://www.techrepublic.com/article/amazon-web-services-lead-the-war-on-cloud-price-reductions>. 2014.
- [16] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven." In: *Proceedings of the Eleventh Symposium on Operating Systems Design and Implementation (OSDI '14)*. Oct. 2014.
- [17] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. "Finding a Needle in Haystack: Facebook's Photo Storage." In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. 2010.
- [18] David Bermbach, Markus Klems, Stefan Tai, and Michael Menzel. "Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs." In: *IEEE International Conference on Cloud Computing (CLOUD '11)*. 2011.
- [19] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. "DepSky: dependable and secure storage in a cloud-of-clouds." In: *ACM Transactions on Storage Systems* 9.4 (2013).
- [20] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. "SCFS: A Shared Cloud-backed File System." In: *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*. 2014.
- [21] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. "Incoop: MapReduce for Incremental Computations." In: *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC '11)*. 2011.
- [22] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. "Single Instance Storage in Windows 2000." In: *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4 (WSS '00)*. 2000.
- [23] Dan Boneh and Richard Lipton. "A Revocable Backup System." In: *Proceedings of the USENIX Security Symposium*. 1996.
- [24] Dan Butcher. *This is how Goldman Sachs is cutting staff through cloud computing*. <https://news.efinancialcareers.com/us-en/263066/this-goldman-md-has-his-head-in-the-cloud-for-a-few-good-reasons>. 2016.

- [25] Brandon Butler. *Amazon speeds up its cloud with SSD block storage*. <http://www.networkworld.com/article/2364506/cloud-storage/amazon-speeds-up-its-cloud-with-ssd-block-storage.html>. 2014.
- [26] *ccache*. <http://ccache.samba.org>.
- [27] Shimin Chen, Michael Kozuch, Theodoros Strigkos, and et.al. “Flexible Hardware Acceleration for Instruction-Grain Program Monitoring.” In: *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 2008.
- [28] Xu Cheng, C. Dale, and Jiangchuan Liu. “Statistics and Social Network of YouTube Videos.” In: *International Workshop on Quality of Service (IWQoS ’08)*. 2008.
- [29] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. “Shredding your garbage: Reducing data lifetime through secure deallocation.” In: *Proceedings of the USENIX Security Symposium*. 2005.
- [30] *Clam AntiVirus*. <http://www.clamav.net>.
- [31] IBM Cloud. *IBM Cloud Functions: Platform Architecture*. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-openwhisk_about.
- [32] CommonCrawl. <http://commoncrawl.org>.
- [33] CompTIA. *Trends in Cloud Computing*. <https://www.comptia.org/about-us/newsroom/press-releases/2016/09/27/companies-becoming-more-measured-in-use-of-cloud-computing-options-new-comptia-study-finds>. 2016.
- [34] Comscore. *Comscore Releases January 2014 U.S. Online Video Rankings*. <http://www.comscore.com/Insights/Press-Releases/2014/2/comScore-Releases-January-2014-US-Online-Video-Rankings>. 2014.
- [35] Ken Corless, Mike Kavis, and Kieran Norton. *NoOps in a serverless world*. <https://www2.deloitte.com/insights/us/en/focus/tech-trends/2019/noops-serverless-computing-transforming-it-operations.html>. 2019.
- [36] Mark D. Corner and Brian D. Noble. “Protecting applications with transient authentication.” In: *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 2003.
- [37] Mark D. Corner and Brian D. Noble. “Zero-interaction authentication.” In: *Proceedings of the ACM Annual International Conference on Mobile Computing and Networking*. 2002.

- [38] Landon P. Cox and Peter Gilbert. *RedFlag: Reducing Inadvertent Leaks by Personal Machines*. Tech. rep. TR-2009-02. Duke University, 2009.
- [39] CumulusClips. <http://cumulusclips.org>.
- [40] Dropbox. <https://www.dropbox.com>.
- [41] Dropbox. *Dropbox Fact Sheet*. <https://www.dropbox.com/static/docs/DropboxFactSheet.pdf>.
- [42] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. "HYDRAsTOR: A Scalable Secondary Storage." In: *Proceedings of the 7th Conference on File and Storage Technologies (FAST '09)*. 2009.
- [43] Laura DuBois, Marshall Amaldas, and Eric Sheppard. *Key considerations as deduplication evolves into primary storage*. White Paper 223310. Mar. 2011.
- [44] Bertrand Dufrasne, Peter Kimmel, Matthew Houzenga, and Dennis Robertson. *IBM DS8000 Easy Tier*. <https://www.redbooks.ibm.com/abstracts/redp4667.html>.
- [45] Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. "Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels." In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2012.
- [46] Elgg. <http://www.elgg.org>.
- [47] EMC. *Fully Automated Storage Tiering (FAST)*. <http://www.emc.com/corporate/glossary/fully-automated-storage-tiering.htm>.
- [48] EncFS. <https://www.arg0.net/encfs>. 2010.
- [49] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." In: *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.
- [50] Eric Enge. *Where is the Mobile vs. Desktop Story Going?* <https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study>. 2019.

- [51] Facebook. <http://www.facebook.com>.
- [52] Federal Communications Commission. *Announcement of New Initiatives to Combat Smartphone and Data Theft*. <https://www.fcc.gov/document/announcement-new-initiatives-combat-smartphone-and-data-theft>. 2012.
- [53] FFmpeg. <https://ffmpeg.org>.
- [54] FileSender. <http://www.filesender.org>.
- [55] *Flexible I/O Tester*. <https://github.com/axboe/fio>.
- [56] Flickr. <http://www.flickr.com>.
- [57] Cloud Foundry. *Where PaaS, Containers and Serverless Stand in a Multi-Platform World*. <https://www.cloudfoundry.org/multi-platform-trend-report-2018>. 2018.
- [58] FUSE: Filesystem in Userspace. <http://fuse.sourceforge.net>.
- [59] Future of Privacy Forum, Center for Democracy & Technology. *Best Practices for Mobile Applications Developers*. <http://www.futureofprivacy.org/wp-content/uploads/Apps-Best-Practices-v-beta.pdf>. 2011.
- [60] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. “Keypad: An auditing file system for theft-prone devices.” In: *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. 2011.
- [61] Roxana Geambasu, Tadayoshi Kohno, Amit Levy, and Henry M. Levy. “Vanish: Increasing Data Privacy with Self-Destructing Data.” In: *Proceedings of the USENIX Security Symposium*. 2009.
- [62] *Goofys: a high-performance, POSIX-ish Amazon S3 file system written in Go*. <https://github.com/kahing/goofys>.
- [63] Google. *Cloud Functions Execution Environment*. <https://cloud.google.com/functions/docs/concepts/exec>.
- [64] Google. *Google Chrome OS*. <http://www.google.com/chromeos/index.html>.
- [65] Google. *MonkeyRunner*. https://developer.android.com/tools/help/monkeyrunner_concepts.html. 2012.
- [66] Google. *Protocol Buffers*. <https://developers.google.com/protocol-buffers>.

- [67] Google. *Two-step verification*. <https://support.google.com/accounts/bin/topic.py?hl=en&topic=28786>. 2012.
- [68] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. "Cost Effective Storage Using Extent Based Dynamic Tiering." In: *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*. 2011.
- [69] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters." In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. 2010.
- [70] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. "Lest We Remember: Cold Boot Attacks on Encryption Keys." In: *Proceedings of the USENIX Security Symposium*. 2008.
- [71] Allan Heydon, Roy Levin, and Yuan Yu. "Caching Function Calls Using Precise Dependencies." In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDIL '00)*. 2000.
- [72] HP. *HPE 3PAR StoreServ 7000 Storage*. <http://www8.hp.com/us/en/products/disk-storage/product-detail.html?oid=5335712>.
- [73] Yuchong Hu, Henry CH Chen, Patrick PC Lee, and Yang Tang. "NCCloud: applying network coding for the storage repair in a cloud-of-clouds." In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*. 2012.
- [74] IBM. *Composer*. <https://github.com/ibm-functions/composer>.
- [75] Imperva. *Consumer Password Practices*. https://www.imperva.com/docs/WP_Consumer_Password_Worst_Practices.pdf. 2010.
- [76] Intel. *Intel Trusted Execution Technology: White Paper*. <http://www.intel.com/content/www/us/en/trusted-execution-technology/trusted-execution-technology-security-paper.html>.
- [77] Intel. *Laptop Security with Intel Anti-Theft Technology*. <http://www.intel.com/content/www/us/en/architecture-and-technology/anti-theft/anti-theft-general-technology.html>. 2012.
- [78] Intel. *Software Guard Extensions Programming Reference*. <https://software.intel.com/sites/default/files/329298-001.pdf>.

- [79] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. “Oozie: Towards a Scalable Workflow Management System for Hadoop.” In: *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies (SWEET '12)*. 2012.
- [80] Adrienne Jeffries. *The man behind Flickr on making the service 'awesome again'*. <http://www.theverge.com/2013/3/20/4121574/flickr-chief-markus-spiering-talks-photos-and-marissa-mayer>. 2013.
- [81] Apache Kafka. *A distributed streaming platform*. <https://kafka.apache.org>.
- [82] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. “Pocket: Elastic Ephemeral Storage for Serverless Analytics.” In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 2018.
- [83] H. Krawczyk and P. Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. tools.ietf.org/html/rfc5869. 2010.
- [84] Kristina Lerman and Laurie A. Jones. “Social Browsing on Flickr.” In: *International Conference on Weblogs and Social Media*. 2007.
- [85] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. “CloudCmp: Comparing Public Cloud Providers.” In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. 2010.
- [86] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. “Architectural Support for Copy and Tamper Resistant Software.” In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2000.
- [87] Yanhong A. Liu, Scott D. Stoller, and Tim Teitelbaum. “Static Caching for Incremental Computation.” In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.3 (May 1998), pp. 546–585.
- [88] Lookout Mobile Security. *Lost and Found: The Challenges of Finding Your Lost or Stolen Phone*. <http://blog.mylookout.com/blog/2011/07/12/lost-and-found-the-challenges-of-finding-your-lost-or-stolen-phone>. 2011.
- [89] Philip MacKenzie and Michael K. Reiter. “Networked cryptographic devices resilient to capture.” In: *Proceedings of the USENIX Security Symposium*. 2001.
- [90] Udi Manber. “Finding Similar Files in a Large File System.” In: *Proceedings of the USENIX Winter 1994 Technical Conference (WTEC '94)*. 1994.

- [91] Lucas Mearian. *World's data will grow by 50X in next decade, IDC study predicts*. http://www.computerworld.com/s/article/9217988/World_s_data_will_grow_by_50X_in_next_decade_IDC_study_predicts. 2011.
- [92] Dutch T. Meyer and William J. Bolosky. "A Study of Practical Deduplication." In: *ACM Transactions on Storage (TOS)* 7.4 (Jan. 2012), 14:1–14:20.
- [93] Donald Michie. "'Memo' Functions and Machine Learning." In: *Nature* 218 (Apr. 1968), pp. 19–22.
- [94] Microsoft. *About Durable Functions*. <https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview>.
- [95] Microsoft. *Create strong passwords*. <http://www.microsoft.com/security/online-privacy/passwords-create.aspx>. 2012.
- [96] Microsoft. *Windows 7 BitLocker Executive Overview*. [https://technet.microsoft.com/en-us/library/dd548341\(WS.10\).aspx](https://technet.microsoft.com/en-us/library/dd548341(WS.10).aspx). 2009.
- [97] Mark Milian. *U.S. government, military to get secure Android phones*. <https://www.cnn.com/2012/02/03/tech/mobile/government-android-phones/index.html>. 2012.
- [98] MiniMatters. *The Best Video Length for Different Videos on YouTube*. <http://www.minimatters.com/blog/youtube-best-video-length>.
- [99] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. "A Low-Bandwidth Network File System." In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. 2001.
- [100] Jacob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.
- [101] NumPy. <https://www.numpy.org>.
- [102] OpenCV. <https://opencv.org>.
- [103] Apache OpenWhisk. *Open Source Serverless Cloud Platform*. <https://openwhisk.apache.org>.
- [104] Thanasis G. Papaioannou, Nicolas Bonvin, and Karl Aberer. "Scalia: An Adaptive Scheme for Efficient Multi-cloud Storage (SC '12)." In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012.
- [105] *Parallel BZIP2 (PBZIP2)*. <http://compression.ca/pbzip2>. 2011.

- [106] Radia Perlman. “File System Design with Assured Delete.” In: *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*. 2007.
- [107] Peter A. H. Peterson. “Cryptkeeper: Improving security with encrypted RAM.” In: *Proceedings of the IEEE International Conference on Technologies for Homeland Security (HST)*. 2010.
- [108] Pillow. <https://pillow.readthedocs.io/en/stable>.
- [109] Piwigo. <http://piwigo.org>.
- [110] Ponemon Institute. *The Lost Smartphone Problem*. <https://www.mcafee.com/us/resources/reports/rp-ponemon-lost-smartphone-problem.pdf>. 2011.
- [111] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. “CryptDB: Protecting Confidentiality with Encrypted Query Processing.” In: *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 2011.
- [112] Niels Provos. “Encrypting virtual memory.” In: *Proceedings of the USENIX Security Symposium*. 2000.
- [113] *Public Data Sets*. <http://aws.amazon.com/datasets>.
- [114] William Pugh and Tim Teitelbaum. “Incremental Computation via Function Caching.” In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’89)*. 1989.
- [115] Krishna PN Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. “Frugal storage for cloud file systems.” In: *Proceedings of the 2012 ACM European Conference on Computer Systems (EUROSYS ’12)*. 2012.
- [116] Redis. <http://redis.io>.
- [117] Jordan Robertson. *Security chip that does encryption in PCs hacked*. https://www.usatoday.com/tech/news/computersecurity/2010-02-08-security-chip-pc-hacked_N.htm. 2010.
- [118] s3fs. <https://github.com/s3fs-fuse/s3fs-fuse>.
- [119] s3ql. <https://bitbucket.org/nikratio/s3ql>.
- [120] Michael Savage. *NHS ‘loses’ thousands of medical records*. <https://www.independent.co.uk/news/uk/politics/nhs-loses-thousands-of-medical-records-1690398.html>. 2009.

- [121] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *Proceedings of the USENIX Annual Technical Conference (USENIX '12)*. 2012.
- [122] Amazon Web Services. *AWS Lambda Developer Guide: Programming Model*. <https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html>.
- [123] Amazon Web Services. *AWS Step Functions*. <https://aws.amazon.com/step-functions>.
- [124] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems." In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*. 2005.
- [125] E. Shi, A. Perrig, and L. van Doorn. "BIND: a fine-grained attestation service for secure distributed systems." In: *IEEE Symposium on Security and Privacy (SP '05)*. 2005.
- [126] Christopher Soghoian. *How Dropbox sacrifices user privacy for cost savings*. <http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html>. 2011.
- [127] Richard P. Spillane, Pradeep J. Shetty, Erez Zadok, Sagar Dixit, and Shrikar Archak. "An Efficient Multi-tier Tablet Server Storage Architecture." In: *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC '11)*. 2011.
- [128] Ahren Studer and Adrian Perrig. "Mobile User Location-specific Encryption (MULE): Using Your Office as Your Password." In: *Proceedings of the ACM Conference on Wireless Network Security (WiSec)*. 2010.
- [129] Symantec Corporation. *PGP Whole Disk Encryption*. <https://www.symantec.com/whole-disk-encryption>. 2012.
- [130] Byung Chul Tak, Bhuvan Urgaonkar, and Anand Sivasubramaniam. "To Move or Not to Move: The Economics of Cloud Computing." In: *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud '11)*. 2011.
- [131] Yang Tang, Patrick P. C. Lee, John C. S. Lui, and Radia Perlman. "FADE: Secure overlay cloud storage for file assured deletion." In: *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*. 2010.

- [132] Radu Timofte, Eirikur Agustsson, Luc Van Gool, Ming-Hsuan Yang, Lei Zhang, Bee Lim, et al. “NTIRE 2017 Challenge on Single Image Super-Resolution: Methods and Results.” In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. 2017.
- [133] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. “HydraFS: A High-throughput File System for the HYDRASor Content-addressable Storage System.” In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST ’10)*. 2010.
- [134] Michael Vrabie, Stefan Savage, and Geoffrey M. Voelker. “BlueSky: A Cloud-backed File System for the Enterprise.” In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST ’12)*. 2012.
- [135] Michael Vrabie, Stefan Savage, and Geoffrey M. Voelker. “Cumulus: Filesystem Backup to the Cloud.” In: *Trans. Storage* 5.4 (Dec. 2009), 14:1–14:28.
- [136] W3C. *Mobile App Best Practices*. <https://www.w3.org/TR/mwabp>. 2010.
- [137] Kevin Walsh and Emin Gün Sirer. “Experience with an Object Reputation System for Peer-to-peer Filesharing.” In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3 (NSDI ’06)*. 2006.
- [138] Hui Wang and Peter Varman. “Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-aware Allocation.” In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST ’14)*. 2014.
- [139] Hal Wasserman and Manuel Blum. “Software Reliability via Run-time Result-checking.” In: *Journal of the ACM (JACM)* 44.6 (Nov. 1997), pp. 826–849.
- [140] Webopedia. *Enterprise storage*. http://www.webopedia.com/TERM/E/enterprise_storage.html.
- [141] Alma Whitten and J.D. Tygar. “Why Johnny can’t encrypt: A usability evaluation of PGP 5.0.” In: *Proceedings of the USENIX Security Symposium*. 1999.
- [142] Alexander Wieder, Pramod Bhatotia, Ansley Post, and Rodrigo Rodrigues. “Orchestrating the Deployment of Computations in the Cloud with Conductor.” In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI ’12)*. 2012.
- [143] Wikimedia. *Media storage*. https://wikitech.wikimedia.org/wiki/Media_storage.

- [144] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. "SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services." In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*. 2013.
- [145] *yextend*. <https://github.com/BayshoreNetworks/yextend>.
- [146] YouTube. <http://www.youtube.com>.
- [147] YouTube. *Statistics - Youtube*. <https://www.youtube.com/yt/press/statistics.html>.
- [148] Jia Yu, Rajkumar Buyya, and Kotagiri Ramamohanarao. "Workflow Scheduling Algorithms for Grid Computing." In: *Metaheuristics for Scheduling in Distributed Computing Environments*. Ed. by Fatos Xhafa and Ajith Abraham. Springer Berlin Heidelberg, 2008, pp. 173–214.
- [149] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language." In: *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*. 2008.
- [150] *ZFS: the last word in file systems*. <http://www.sun.com/2004-0914/feature>.
- [151] Gong Zhang, Lawrence Chiu, and Ling Liu. "Adaptive data migration in multi-tiered storage based cloud environment." In: *IEEE 3rd International Conference on Cloud Computing (CLOUD '10)*, 2010.
- [152] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert Dick, Z. Morley Mao, and Lei Yang. "Accurate online power estimation and automatic battery behavior based power model generation for smartphones." In: *Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 2000.