Title
Protecting User Privacy in Remotely Managed Applications

Author
Mohan, Prashanth

Publication Date
2013

Peer reviewed|Thesis/dissertation

Protecting User Privacy in Remotely Managed Applications

by

Prashanth Mohan

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division
of the
University of California, Berkeley

Committee in charge:
Professor David E. Culler, Chair
Professor John Chuang
Professor Dawn Song

Fall 2013

Protecting User Privacy in Remotely Managed Applications

Abstract

Protecting User Privacy in Remotely Managed Applications

by

Prashanth Mohan
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Chair

This thesis presents an end-to-end system architecture for online services to provide it's users with a "privacy guarantee". The privacy guarantee as described in this thesis relates to the technological enforcement of the user's privacy policies by these online applications that are otherwise untrusted by the user.

Applications on the Internet are complex in that they integrate different types of functionalities into a consistent interface for the user. This thesis categorizes these functionalities into three generic components – a learning module that operates on the data from multiple users to gather higher level trends and aggregates, a data storage and transformation module that provides the core functionality of data presentation and finally a client-side component that interacts with the cloud-side functionalities and is responsible for sourcing input from the user and presenting them on the user's device in a secure fashion.

This thesis looks at the privacy risks introduced by each of these components and describes a "trusted system" that can be used by these online services to prove that the user specified privacy policies are enforced. The system consists of multiple independently developed solutions – Gupt, Rubicon, Bubbles and MobAds. These solutions work at tandem with each other to provide an end-to-end privacy perspective.

While privacy policies and EULAs have largely been enforced in the realm of legal proceedings, this prototype implementation of an end-to-end privacy enforcement architecture demonstrates that it is both feasible and practical to enforce user privacy policies within the system.

To everyone who has believed in me, especially my family.

Contents

List of Figures

List of Tables

Acknowledgements

I deeply indebted to my Advisor David Culler whose valuable guidance, enthusiasm, and encouragement has helped steer my projects in many interesting directions. I am also extremely thankful to Dawn Song who has been instrumental in defining most of the projects defined in this thesis. The wonderful feedback I received from my other qualification committee members – Professors Anthony Joseph and John Chuang has also been critical to the positioning of my work in relation to real world practical demands.

I would also like to acknowledge the invaluable contributions of my colleagues and collaborators Elaine Shi, Abhradeep Thakurta, Emil Stefanov, Mohit Tiwari, Ch. Papamanthou, David Zats, Tathagatha Das, Andrew Krioukov, Stephen Dawson-Haggerty, Jorge Ortiz, Sara Alspaugh, Jeff Hsu, Xiaofan Jiang, Noah Johnson, Adrian Mettler, Hui Xue, and Peter Gilbert.

I would like to especially point out the significance of my mentors – Nagappan Alagappan, Venkat Padmanabhan, Ramachandran Ramjee, Oriana Riva, Suman Nath, Venkatesh Kancharla, Viswanath Sankaranarayanan and Raymond Wei at Novell Inc, Microsoft Research, Amazon.com and FireEye during my time in the industry.

Several chapters in this dissertation have benefited from the generous efforts of colleagues providing feedback on earlier drafts. I thank David Zats, Ganesh Ananthanarayanan, Aurojit Panda, Arka Bhattacharya, Shivaram Venkataraman and others I have unfortunately missed mentioning for providing me valuable feedback on papers and presentations. It is the constant prodding by these friends that took me past the finish line. I also thank shepherds Steve Hand and Sam Madden, and the many anonymous conference reviewers for their helpful feedback on submitted papers.

Finally and most importantly, this thesis would never have become a reality without the love and support of my family, who have always encouraged me to pursue my dreams.

Chapter 1

Introduction

Alan Westin [141] very eloquently described privacy as:

> ... the claim of individuals, groups, or institutions to determine for themselves when, how, and to what extent information about them is communicated to others.

This thesis adopts this definition of privacy and studies the risks that a user assumes when sharing his or her information with online services. In today's internet ecosystem, online businesses are developed around the model of monetizing the user's information and often pass on the costs of the service to the merchants in exchange for user information, rather than charge the user. There is thus a clear separation between the user (the individual partaking in the online service) and the customer (the merchant seeking to sell to that user). This inevitably leads to the user accepting some privacy risks in exchange for the service. This thesis works within this framework and makes these privacy risks explicit to the user without sacrificing current business models. It proposes technological solutions that provide these services with the means to allow users to set boundaries or limits on the set of principals who have access to their data. In return, the services can provide to the user a technologically enforced "privacy guarantee" that the user's privacy expectations of the online service are met.

1.1 Problem Statement

When using an application, the user today lays implicit trust on the application not being actively malicious. Further, the user also has to trust that the application developers and infrastructure providers will make judicious decisions to protect the user's privacy, including hardening their operating systems, promptly applying security updates, and using appropriate authentication, encryption and information flow control mechanisms correctly. However, benign developers regularly skip best practices for secure application development. Security misconfigurations are so prevalent that it has been rated as the 6th most dangerous

web application vulnerability and risk to organizations [106]. Worse, a malicious developer can deliberately misuse her overarching access to users' data [133] or circumvent existing permission-based mechanisms [24] to compromise users' privacy. Thus, even if an application advertises desirable features, users are often unwilling to use applications from unknown developers. Ideally, the task of specifying and enforcing privacy rules should be separated from individual applications and developers; it should be done *once* for *all* applications by a trusted underlying system.

The diversity in applications requires the innovative use of user data to increase user satisfaction. For instance, social networks have enabled applications using social data to provide services such as gaming, federated logins, online coupons, car-pooling, etc. Similarly, retailers share data about their customers' purchasing behavior with product merchants for more effective and targeted advertising. While some applications store, process and deliver a single user's data, other applications perform machine learning and other analytics on data from multiple users in order to improve the experience of individual users. Organizations also frequently allow third parties to perform business analytics and provide services using aggregated data.

While sharing information can be highly valuable, companies severely restrict access to user data because of the risk that an individual's privacy would be violated. Laws such as the *Health Insurance Portability and Accountability Act (HIPAA)* have considered the dangers of privacy loss in health data and stipulate that patient's personally identifiable information (PII) should not be shared.

To overcome these privacy issues, organizations have resorted to data transformation techniques that attempt to anonymize the users in the dataset. Unfortunately, even in datasets with "anonymized" users, there have been cases where user privacy was breached. Examples include the deanonymization of AOL search logs [14], the identification of patients in a Massachusetts hospital by combining the public voters list with the hospital's anonymized discharge list [124] and the identification of the users in an anonymized Netflix prize data using an IMDB movie rating dataset [102].

The central question this thesis explores is:

> **Users manage their data on remotely controlled services that allows untrusted applications to consume this in order to provide the user with useful services. How can we design a technological solution that enforces the user-specified privacy policies on this dynamic system with multiple participating untrusted entities?**

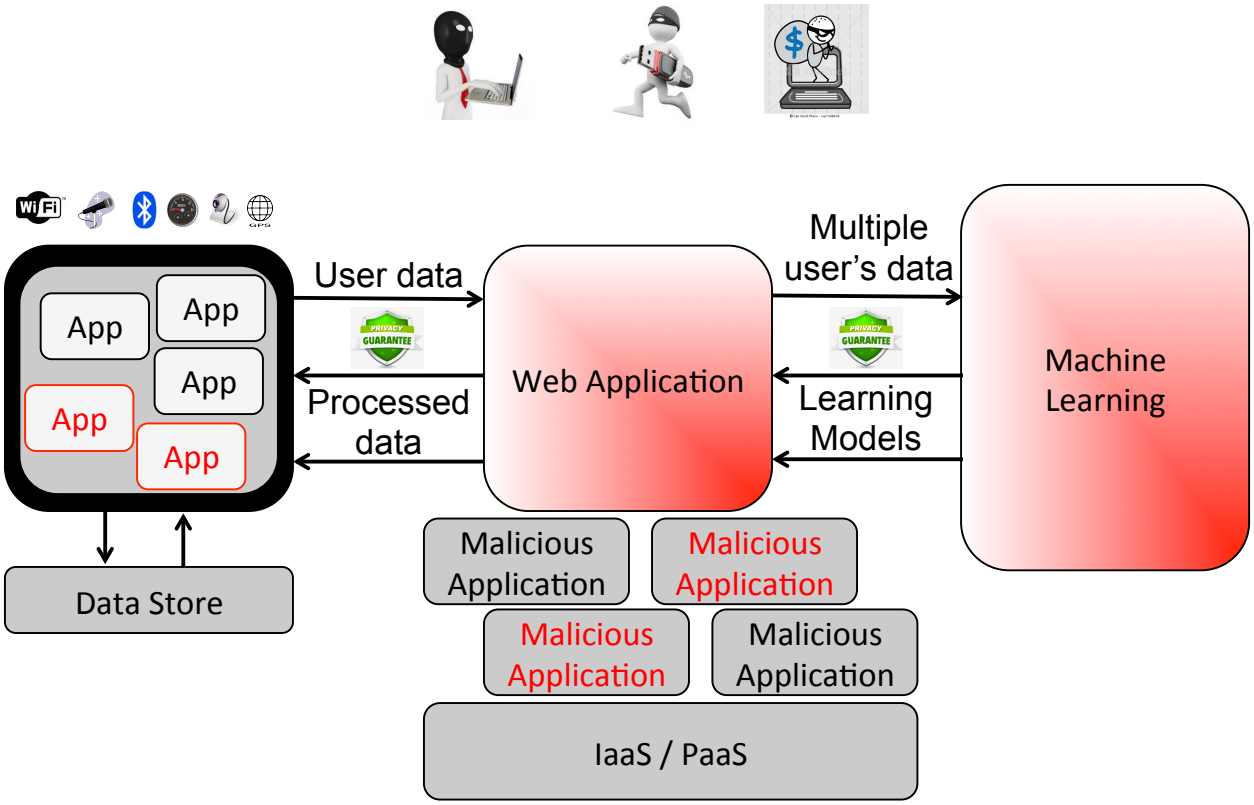


Figure 1.1: A holistic view of the attack surface and threat vectors for a user of a cloud-based data management system.

1.2 An end-to-end privacy preserving service architecture

This thesis describes the architecture of a information flow system for cloud-based data management solutions. We identify the risks in data leakage at each of the components. Specifically, this thesis addresses the privacy challenges in the following three categories:

Hiding in the crowd: An analyst performing large scale data analysis on users' data should not be able to determine the personally identifying information of any single individual in the dataset.

Limiting access to information: Users should be able to explicitly restrict and permit various parties that have access to their personally identifying information.

Trusting your own device: An integral part of any remote service is the local device through which the user accesses online services and feeds in the sensitive information.

Moving functionality to the end host partially alleviates some of the privacy concerns. It is however, still feasible for online vendors to surreptitiously steal the users' information.

Side channel attacks: In the cloud when applications are often hosted on co-located environments, it is possible that the application instance would be on the same physical host as a different application from an attacker. The attacker can attempt to extract information through side channels such as timing information. This category of risks are not discussed in this thesis.

Online applications are extremely complex and contain various components that exhibit diverse capabilities. We attempt to generalize these functionalities into the following components:

Data aggregator: These components aggregates information from a number of users in order to draw statistical analyses. These components offer the risk that the output of the analysis system might maliciously or inadvertently expose the personally identifying information of specific individuals in the dataset. This directly relates to the first category of privacy challenges.

Siloed data morphing: These components work on the data for individual users – storing and transforming them into information that is palatable to the user. The privacy risk introduced by these components is more straightforward. Note that the same instance of the component would typically serve multiple users at the same time. There now is the channel by which the component being semi-trusted might pass on information of one individual to a different user. This lends itself directly to the second category of privacy risk – that of restricting information access.

On-device application interface: The security of the system is only as secure as the weakest link. It is often the case that the end-client devices are less actively managed and exposed to different adversarial scenarios. It is thus imperative that the device have a well defined and verifiable trusted code base with strong data isolation enforcement based on user specified privacy policies. The interface of the application can be as simple as a browser based representation to a more complex native application.

This thesis assumes that all of these different components are either untrusted of semi-trusted. A typical online service consists of all of these components. The application interface is used to source the data fed to the web application. The application aggregates the data from multiple users and runs statistical models on this data set to garner higher level trends. The application uses this information to transform the user's data and represent it a more convenient model.

1.3 Organization

This thesis builds on well studied security concepts including information flow control and differential privacy. Chapter 2 provides a high level description of these mature concepts and their application to the problem of preserving user privacy in online services. Subsequent chapters in this thesis expand on how these fundamentals are utilized and extended building up to the software architecture for providing users with a privacy guarantee.

This thesis describes mechanisms that provide users with a privacy guarantee while allowing untrusted applications to analyze the user's data and provide services. Chapter 3 dives into the first of the three categories described above. This chapter describes the design and development of GUPT[1] – a framework that provides users with a guarantee that when the user's data is part of a big data analysis, an individual user's information is not exposed. This guarantee is provided while still allowing for the analyst to utilize the aggregate information. Gupt is a data platform that allows organizations to execute data analyses by untrusted third-parties while preserving the privacy of the individuals in the underlying datasets. Rewriting existing programs to be privacy compliant is both expensive and demanding even of programmers with sophisticated mathematical skills. Gupt overcomes this obstacle and allows the execution of existing programs with no modifications, eliminating the expensive and demanding task of rewriting programs to be differentially private. Gupt also allows data analysts to specify a desired output accuracy rather than work with the abstract notion of a privacy budget. Finally, Gupt automatically parallelizes the task across a cluster ensuring scalability for concurrent analytics.

Chapter 4 describes RUBICON[2] – a trusted *platform* that allows users to run untrusted applications on their sensitive data while maintaining end-to-end privacy constraints of the user and still preserving the functionality of the application. The main objective of Rubicon is to eliminate trust on the cloud application developer and allow users to execute third-party applications while still ensuring that the user's privacy policies are enforced, yet allowing for a rich and fulfilling experience. Applications are thus considered to be untrusted in this setting and could actively try to breach privacy policies defined by the user. For example, they could attempt to leak data by connecting to arbitrary web domains or by mixing data between users. We also assume that *users* are responsible for choosing with who to share their data[3]. Both Rubicon and Gupt can be used either in conjunction or independently.

The lack of willingness of users (especially enterprise users) to adopt applications that store data in the cloud, organizations are starting to provide disconnected services that continue to store data on the end host device while delivering applications from the cloud. With the growth in data theft, this approach is also increasingly becoming popular among the online service provider as a means to reduce their liability. Chapter 5 extends and adapts

[1] Gupt is a Sanskrit word meaning 'Secret'.

[2] Rubicon is a river in Italy, that was key to protecting Rome from civil war and important for Caesar's crossing of it.

[3] A user may turn around and republish another user's information to the whole world; defense against such attacks (i.e., the "analog hole") is not analyzed in this thesis.

the data isolation concepts discussed in Chapter 4 to end host applications. An entirely disconnected mode of operation however does not provide users with all of the functionalities that a cloud-based operation enables. Chapter 5 also discusses mechanisms to preserve user privacy in a hybrid mode of operation wherein the application only shares limited data to the cloud service through anonymized communication channels. We specifically scrutinize the ad-delivery mechanism as an example. This chapter includes a study of the ad delivery architecture and a system that delivers customized ads to users while maintaining the real time guarantees that are necessary for online advertising systems.

Finally, Chapter 6 summarizes the types of privacy risks and enforcement mechanisms for online applications that were studied in the previous chapters.

1.4 Contributions

This thesis describes in depth a number of independently developed systems – Gupt in Chapter 3, Rubicon in Chapter 4, Bubbles and MobAds in Chapter 5. Each of these system individually make technical contributions that when put together provide us with the end-to-end privacy properties.

Overall, the thesis makes the following contributions:

- Describes the design and development of the Gupt framework that ensures that the output of unmodified applications run on sensitive data sets is always differentially private.
- Introduces the idea of an aging of sensitivity that allows Gupt to optimize various parameters of the system allowing for data analysts (the users) to only specify the expected accuracy of the output and not worry about the mathematical privacy budget construct.
- Proposes a novel software design pattern called Application-Storage-Template (AST) which is an extension of the traditional multi-tiered systems. This pattern allows for the transparent translation of access control lists to information flow control rules.
- Rubicon provides a practical and efficient system that executes AST applications and provides users with a privacy guarantee that their data always conforms to the user-specified privacy policies even if it is accessed by untrusted applications.
- Bubbles offers a new way to consolidate and manage the privacy of user data through mobile smartphones based on real life contextual information.
- For the applications such as online advertisements that require dis-connected activity to preserve user privacy, MobAds describes a system that allows prefetching of content without affecting the financial incentives of the online service provider.

1.5 Statement of joint work

This thesis draws heavily from the reports published about Gupt [98], Rubicon, Bubbles [136] and MobAds [97].

I am the primary developer of the Gupt system and maintain the code at `http://github.com/prashmohan/GUPT/`. It was designed along with my collaborators - Abhradeep Thakurta, Elaine Shi, Dawn Song and David Culler. The Rubicon system was designed along with my collaborators - Emil Stefanov, Mohit Tiwari, Ngyuen Tran, Charalampos Papamanthou, Jin Chen, Petros Maniatis, Elaine Shi, Dawn Song, Krste Asanovi and David Culler. The primary development of this system was led by Emil Stefanov and myself. The Bubbles system was designed in collaboration with Mohit Tiwari, Andrew Osheroff, Hilfi Alkaff, Elaine Shi, Eric Love, Dawn Song and Krste Asanovi. I was involved in the conception and design of this project while the primary development was led by Andrew Osheroff and Mohit Tiwari. I led the design and development of the MobAds system during an summer internship at Microsoft Research under the guidance of Oriana Riva and Suman Nath.

Chapter 2

Background

2.1 Data anonymization

The value in sharing data is often obtained by allowing analysts to run aggregate queries spanning a large number of entities in the dataset while disallowing analysts from being able to extract data that pertains to individual entities. For instance, a merchant performing market research to identify their next product would want to analyze customer behavior in retailers' databases. The retailers might be willing to monetize the dataset and share aggregate analytics with the merchant, but would be unwilling to allow the merchant to extract information specific to individual customers in the database.

Researchers have invented techniques that ranged from ad-hoc obfuscation of data entries (such as the removal of Personally Identifiable Information) to more sophisticated anonymization mechanisms satisfying privacy definitions like k-anonymity [134] and ℓ-diversity [84]. However, Ganta *et al.* [42] and Kifer [65] showed that practical attacks can be mounted against these techniques. *Differential privacy* [31] is a definition of privacy that formalizes the notion of privacy of an individual in a dataset.

Unlike earlier techniques, differentially private mechanisms use statistical techniques that allow data owners to explicitly specify bounds on the amount of information that can be extracted from the dataset. This means that an adversary will end up with just as much information as he or she had before attempting to extract information about specific entities in the dataset. For example, during the NetFlix challenge the authors of [102] deanonymized specific users of NetFlix by using a combination of the publicly available IMDB data with the anonymized NetFlix usage data (more examples of such deanonymization using multiple datasets are available in [19,85]). In such cases, if access to the sensitive dataset is restricted to differential privacy techniques with strict restriction on the amount of privacy that can be leaked, these forms of user identification can be avoided. Differential privacy can be achieved by perturbing the result of a computation in a manner that has little effect on aggregates, yet obscures the data of individual constituents.

Differential privacy has strong theoretical properties, but the shortcomings of existing

differentially private data analysis systems have limited its adoption. For instance, existing programs cannot be leveraged for private data analysis without modification. The magnitude of the perturbation introduced in the final output is another cause of concern for data analysts. Differential privacy systems operate using an abstract notion of privacy, called the 'privacy budget'. Intuitively a lower privacy budget implies better privacy. However, this unit of privacy does not easily translate into the utility of the program and is thus difficult for data analysts or application developers to interpret. Further, analysts would also be required to efficiently distribute this limited privacy budget between multiple queries operating on a dataset. An inefficient distribution of the privacy budget would result in inaccurate data analysis and reduce the number of queries that can be safely performed on the dataset.

2.2 Differential privacy

Differential privacy places privacy research on a firm theoretical foundation. It guarantees that the presence or absence of a particular record in a dataset will not significantly change the output of any computation on a statistical dataset. An adversary thus learns approximately the same information about any individual record, irrespective of its presence or absence in the original dataset.

Definition 1 (ϵ-differential privacy [31])**.** *A randomized algorithm $\mathcal{A}$ is ϵ-differentially private if for all datasets $T, T' \in \mathcal{D}^n$ differing in at most one data record and for any set of possible outputs $\mathcal{O} \subseteq Range(\mathcal{A})$, $\Pr[\mathcal{A}(T) \in \mathcal{O}] \leq e^{\epsilon} \Pr[\mathcal{A}(T') \in \mathcal{O}]$. Here $\mathcal{D}$ is the domain from which the data records are drawn.*

The privacy parameter ϵ, also called the *privacy budget* [90], is fundamental to differential privacy. Intuitively, a lower value of ϵ implies stronger privacy guarantee and a higher value implies a weaker privacy guarantee while possibly achieving higher accuracy.

2.3 Sample and Aggregate framework

The "Sample and Aggregate" framework [105, 130] (SAF) was originally conceived to achieve optimal convergence rates for differentially private statistical estimators. Given a statistical estimator $\mathcal{P}(T)$, where T is the input dataset , SAF constructs a differentially private statistical estimator $\hat{\mathcal{P}}(T)$ using $\mathcal{P}$ as a black box. Moreover, theoretical analysis guarantees that the output of $\hat{\mathcal{P}}(T)$ converges to that of $\mathcal{P}(T)$ as the size of the dataset T increases.

As the name "Sample and Aggregate" suggests, the algorithm first partitions the dataset into smaller subsets; i.e.,, $\ell = n^{0.4}$ blocks (call them $T_1, \cdots, T_\ell$) (see Figure 1). The analytics program $\mathcal{P}$ is applied on each of these datasets T_i and the outputs O_i are recorded. The O_i's are now clamped to within an output range that is either provided by the analyst or

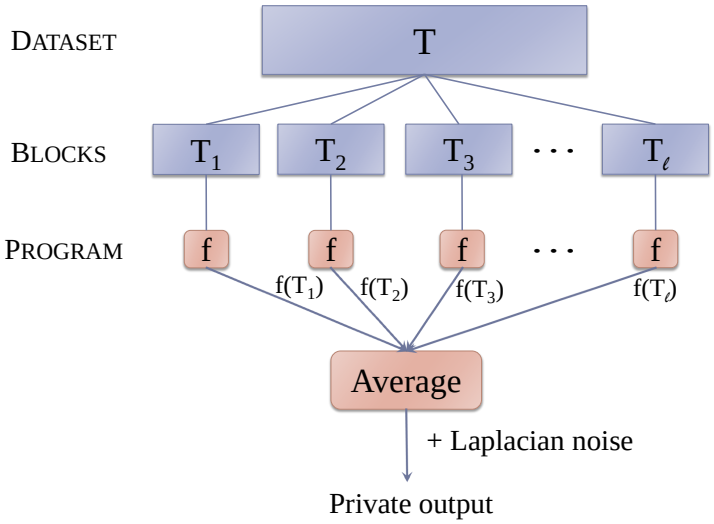


Figure 2.1: An instantiation of the Sample and Aggregate Framework [130].

inferred using a range estimator function. (Refer to Section 3.3.1 for more details.) Finally, a differentially private average of the O_i's is calculated by adding Laplace noise (scaled according to the output range). This noisy final output is now differentially private. The complete algorithm is provided in Algorithm 1. Note that the choice of number of blocks $\ell = n^{0.4}$ is from [130], used here for completeness. For improved choices of ℓ, see Section 3.5.

Algorithm 1 Sample and Aggregate Algorithm [130]

Input: Dataset $T \in \mathbb{R}^n$, length of the dataset n, privacy parameters ϵ, output range $(\min, \max)$.

1: Let $\ell = n^{0.4}$
2: Randomly partition T into ℓ disjoint blocks $T_1, \cdots, T_\ell$.
3: **for** $i \in \{1, \cdots, \ell\}$ **do**
4: $\quad O_i \leftarrow$ Output of user application on dataset T_i.
5: $\quad$ If $O_i > \max$, then $O_i \leftarrow \max$.
6: $\quad$ If $O_i < \min$, then $O_i \leftarrow \min$.
7: **end for**
8: $A \leftarrow \frac{1}{\ell} \sum_{i=1}^{\ell} O_i + \mathsf{Lap}(\frac{|\max - \min|}{\ell \cdot \epsilon})$

2.4 Information flow control

The principle of information-flow control has been studied since the introduction of the traditional security models such as Bell and LaPadula [15] and Denning *et al.* [28]. The *decentralized model* of this topic has also received a significant amount of interest [99]. In this model the propagation of privilege and release of information can be defined by the end users of the applications (and the data), and need not only emanate from a super user.

Information flow has been studied in the contexts of language-based approaches [80,120] (where the developer is required to program in a specialized language suitable for security policies), OS-based approaches [33, 71, 151], and layers in between [72, 118, 129, 149]. Language-based approaches are susceptible to security violations on system resources while OS-based approaches, although controlling accesses to system resources, they cannot monitor flow in more fine-grained program data structures.

Traditionally, DIFC approaches enforce fine-grained information flow policy at the granularity of program variables or files and not necessarily at the granularity (or semantic level) of user-facing data objects. Efstathopoulos and Kohler [32] identified the need for information-flow policy at a semantically higher level, above individual labels and tags; they focused on module-to-module communication policy. Harris *et al.* [52] proceeded along the same path, by providing automated DIFC rewrites.

DIFC systems are only one of many possible way in which untrusted code can be executed in a contained fashion (examples of other mechanisms include program shepherding [67]). The closest in spirit to Rubicon (described in Section 4) is work that attacks what is anecdotally called a *red-green model*, where a trusted 'green' machine operates on sensitive data, and all other data, including untrusted applications, live in a 'red' machine. Sharing between trusted and untrusted applications tends to defeat such red-green approaches in all but the most demanding (i.e., military) applications.

Dynamic Taint Analysis [103] has also been used on unmodified applications, to prevent the disclosure of sensitive data by untrusted code. Unfortunately, most dynamic information-flow tracking solutions are inefficient. Although performance improvements have been achieved through careful engineering [37,153], such approaches still rely on tracking information flow at individual instruction boundaries. Some proposals have addressed tracking only within individual application instances [86].

Chapter 3

Privacy preserving data mining

This chapter studies the privacy risks involved in sharing user data with web services that provide services by mining data from different users'Building on the architecture introduced in Chapter 1, consider a scenario where a service mines data from multiple users. We can view the problem as involving three logical entities:

1. The *analyst/programmer*, who wishes to perform aggregate data analytics over sensitive datasets.

2. The *data owner*, who owns one or more datasets, and would like to allow analysts to perform data analytics over the datasets without compromising the privacy of users in the dataset.

3. The *service provider*, who hosts the Gupt service.

The separation between these three parties is logical; in reality, either the data owner or a third-party cloud service provider could host Gupt. Our goal is to make differentially private analysis easy for an application programmer (who is not a privacy expert) in this context.

Trust assumptions: We assume that the data owner and the service provider are *trusted*, and that the analyst is *untrusted*. In particular, the programs supplied by the analyst may act maliciously and try to leak information. Gupt defends against such attacks using the security mechanisms proposed in Section 3.9.

3.1 Overview and problem setup

Figure 3.1 shows the building blocks of Gupt:

- The *dataset manager* is a database that registers instances of the available datasets and maintains the available privacy budget.

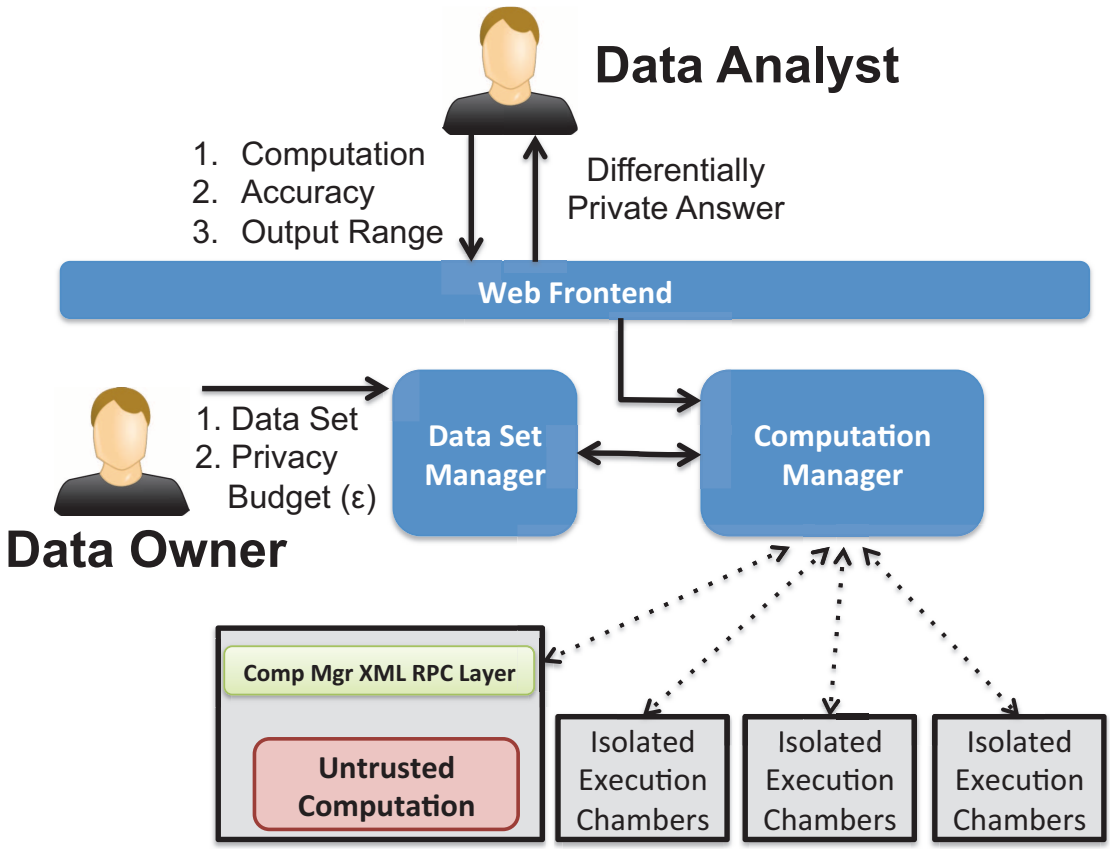


Figure 3.1: Overview of GUPT's Architecture

- The *computation manager* instantiates computations and seamlessly pipes data from the dataset to the appropriate instances.
- The *isolated execution chambers* are POSIX-compatible environments that isolate and prevent any malicious behavior by the computation instances.

These building blocks allow the principals of the system to easily interact with the system parameters either being automatically optimized or optionally over-ridden by experts.

Interface with the data owner: The data owner supplies to Gupt: (a) A multi-dimensional **dataset** (such as a database table) that for the purpose of our discussion, we assume is a collection of real valued vectors, (b) a total **privacy budget** that is allocated for computations on the dataset and (c) *[Optional]* **input attribute ranges**, i.e., the lower and upper bound for each dimension of the data. Section 3.3.1 presents a detailed discussion on these bounds.

For privacy reasons, the input ranges provided should not contain any sensitive information. For example, it is reasonable to expect that a majority of the household's annual income should fall within the range [0; 500,000] dollars, thus it is not considered sensitive.

On the other hand if the richest household's income is used as the upper bound private information could be leaked. In this case, a public information source such as the national GDP could be used.

Interface with the analyst: The data analyst supplies the following to Gupt: (a) **Data analytics program**, (b) a **reference to the data set** in the dataset manager, (c) either a **privacy budget** or the desired **accuracy** for the final answer and finally (d) an **output range** or a **helper function** for estimating the output range.

A key requirement for the analytics program is that it should be able to run on *any* subset of the dataset. Since Gupt executes the application in a black-box fashion, only a binary executable of the program is required.

Privacy budget distribution: In order to guarantee ϵ-differential privacy for a dataset T, the privacy budget should be distributed among the applications (call them $\mathcal{A}_1, \cdots, \mathcal{A}_k$) that operate on T. A *composition lemma* [31] states that if $\mathcal{A}_1, \cdots, \mathcal{A}_k$ guarantee $\epsilon_1, \cdots, \epsilon_k$-differential privacy respectively, then T is ϵ-differential private, where $\epsilon = \sum_{i=1}^{k} \epsilon_i$. Thus judicious allocation of the privacy budget is important. Unlike existing differential privacy solutions, Gupt relieves the analyst and the data owner from that task of distributing this limited privacy budget between multiple data analytics programs, as discussed below.

3.2 Background and related work

Gupt extends the conventional Sample and Aggregate Framework (SAF) described in Chapter 2 in the following ways: i) *Resampling:* Gupt introduces the use of data resampling to improve the experimental accuracy of queries performed using SAF without degrading the privacy guarantee; ii) *Optimal block allocation:* Gupt further improves experimental accuracy by finding the better block sizes (as compared to the default choice of $n^{0.6}$) using the *aging of sensitivity* model explained later in Section 3.4.

A number of advances in differential privacy have sought to improve the accuracy of certain types of data queries, such as linear counting queries [79], graph queries [61] and histogram analysis [53]. A recent system called PASTE [114] allows queries on time series data where the data is stored on distributed nodes and no trust is laid on the central aggregator. In contract to PASTE, Gupt trusts the aggregator with storing all of the data and provides a flexible system that supports many different types of data analysis programs.

While systems tailored for specific tasks could potentially achieve better output accuracy, Gupt trades this for the generality of the platform. We show through experimental results that Gupt achieves reasonable accuracy for problems like clustering and regression, and can even perform better than the existing customized systems.

Other differential privacy systems, such as PINQ [90] and Airavat [119] have also attempted to operate on a wide variety of data queries. PINQ (Privacy INtegrated Queries) proposed programming constructs which enable application developers to write differentially private programs using basic functional building blocks of differential privacy (*e.g.*, *exponential mechanism* [91], *noisy counts* [31] etc.). It *does not* consider the application developer

to be an adversary. It further requires the developers to rewrite the application to make use of the PINQ primitives. On the other hand, Airavat was the first system that attempted to run unmodified programs in a differentially private manner. However, it required the programs to be written for the Map-Reduce programming paradigm [26]. Further, Airavat only considers the map program to be an "untrusted" computation, while the reduce program is "trusted" to be implemented in a differentially private manner. In comparison, Gupt allows for the private analysis of a wider range of unmodified programs. Gupt also introduces techniques that allow data analysts to specify their privacy budget in units of output accuracy. Section 3.11 presents a detailed comparison of Gupt with PINQ, Airavat and SAF.

Similar to iReduct [144], Gupt introduces techniques that reduce the relative error (in contrast to absolute error). Both systems use a smaller privacy budget for programs that produce larger outputs, as the relative error would be small as compared programs that generate smaller values for the same absolute error. While iReduct optimizes the distribution of privacy budget across multiple queries, Gupt matches the relative error to the privacy budget of individual queries.

3.3 Algorithm

The sample and aggregate (SAF) algorithm described in Chapter 2.3 introduces two sources of error:

- *Estimation Error:* This arises because the query is evaluated on smaller data blocks, rather than the entire dataset. Typically, the larger the block size, the smaller the estimation error.

- *Induced Noise:* Another source of error is due to the Laplace noise introduced to guarantee differential privacy.

  Intuitively, the larger the number of blocks, the lower the sensitivity of the aggregation function – since the aggregation function has sensitivity $\frac{s}{\ell}$, where s denotes the output range of each block, and ℓ denotes the number of blocks. As a result, given a fixed privacy parameter ϵ, with a larger number of blocks, the magnitude of the Laplace noise is lowered.

Gupt uses two strategies (*resampling* and *selecting the optimal block size*) to reduce these types of errors. Before delving into details of the techniques, the following section explains how the output range for a given analysis program is computed. This range is used to decide the amount of noise to be added to the final output.

3.3.1 Output range estimation

The sample and aggregate framework described in Algorithm 1 does not describe a mechanism to obtain the range within which the output can lie. This is needed to estimate

the noise that should be added for differential privacy. Gupt implements this requirement by providing the following mechanisms:

1. `GUPT-tight`: The analyst specifies a tight range for the *output*.
2. `GUPT-loose`: The analyst only provides a loose range for the *output*. In this case, the computation is run on each data block and their outputs are recorded. A differentially private percentile estimation algorithm [130] is then applied on the set of outputs to privately compute the 25-th and the 75-th percentile values. These values are used as the range of the output and are supplied to Algorithm 1.
3. `GUPT-helper`: The analyst could also provide a range translation function. If either (a) *input* range is not present or (b) only very loose range for the *input* (e.g., using the national GDP as an upper-bound on annual household income) is available, then Gupt runs the same differentially private percentile estimation algorithm on the inputs to privately compute the 25-th and the 75-th percentile (a.k.a, lower and upper quartiles) of the inputs. This is used as a tight approximation of the input range. The analyst-supplied range translation function is then invoked to convert the "tight" input range into an estimate of the output range.

Our experiments demonstrate that one can get good results for a large class of problems using the noisy lower and upper quartiles as approximations of the output range. If the input dataset is multi-dimensional, the range estimation algorithm is run independently for each dimension. Note that the choice of 25-th and 75-th percentile above is somewhat arbitrary. In fact, one can choose a larger inter-percentile range (*e.g.*, 10-th and 90-th percentile) if there are more data samples. However, this does not affect the asymptotic behavior of the algorithm.

3.3.2 Data resampling

The variance in the final output is due to two sources of randomness: i) partitioning the dataset into blocks and ii) the Laplace noise added. The following resampling technique[1] can be used to reduce the variance due to partitioning the dataset into blocks. Instead of requiring each data entry to reside in exactly one block (as described in the original sample and aggregate framework [130]), each data entry can now reside in multiple blocks. The *resampling factor* γ denotes the number of blocks in which each data entry resides.

If the number of records in the dataset is n, block size is β and each data entry resides in γ blocks, it is easy to see that the number of blocks $\ell = \gamma n/\beta$. To incorporate resampling, we make the following modifications to Algorithm 1. Lines 1 and 2 are modified as follows. Consider $\ell = \gamma n/\beta$ bins of size β each. The i^{th} entry from the dataset T is picked and randomly placed into γ bins that are not full. This process is performed for all the entries

[1] A variant of this technique was suggested by Prof. Adam Smith at University of Pennsylvania.

in the dataset T. In Line 8, the Laplace noise is changed to $\mathsf{Lap}(\frac{\beta|\max-\min|}{n\cdot\epsilon})$. The rest of Algorithm 1 is left intact.

The main benefit of using resampling is that it reduces the variance due to partitioning the dataset into blocks without increasing the noise needed for the same level of privacy.

Claim 1. *With the same privacy level ϵ, resampling with any $\gamma \in \mathbb{Z}^+$, does not increase the Laplace noise being added (for fixed block size β).*

Proof. Since each record appears in γ blocks, a Laplace noise of magnitude $O(\frac{\gamma s}{\epsilon \ell}) = O(\frac{s\beta}{\epsilon n})$ should be added to preserve ϵ-differential privacy. This means that once the block size is fixed, the noise is independent of the factor γ. □

Intuitively, the benefit from resampling is that the variance due to partitioning of the dataset into blocks is reduced without increasing the Laplace noise (added in Step 8 of Algorithm 1) needed for the same level of privacy with the inclusion of $\gamma > 1$. Consider the following example to get a better understanding of the intuition.

Example 1. *Let T be a dataset (with n records) of the ages of a population and max be the maximum age in the dataset. The objective is to find the average age (Av) in this dataset. Let $\hat{Av}$ be the average age of a dataset formed with $n^{0.6}$ uniformly random samples drawn from T with replacement. The expectation of $\hat{Av}$ equals the true average Av. However, the variance of $\hat{Av}$ will not be zero (unless all the entries in T are same). Let $\mathcal{O} = \frac{1}{\psi}\sum_{i=1}^{\psi} \hat{Av}(i)$, where $\hat{Av}(i)$ is the i-th independent computation of $\hat{Av}$ mentioned above and ψ is some constant. Notice that $\mathcal{O}$ has the same expected value as $\hat{Av}$ but the variance has reduced by a factor of ψ. Hence, resampling reduces the variance in the final output $\mathcal{O}$ without introducing bias.*

The above example is a simplified version of the actual resampling process. In the actual resampling process each data block of size $n^{0.6}$ is allowed to have only one copy of each data entry of T. However, even with an inaccurate representation, the above example captures the essence of the underlying phenomenon.

In practice, the resampling factor γ is picked such that it is reasonably large, without increasing the computation overhead significantly. Notice that the increase in accuracy with the increase of γ becomes insignificant beyond a threshold.

3.4 Aging of privacy

In real life datasets, the potential privacy threat for each record is different. A privacy mechanism that considers this can obtain good utility while satisfying the privacy constraints.

We introduce a new model called *aging of sensitivity* of data where older data records are considered to have lower privacy requirements. Gupt uses this model to optimize some parameters of the sample and aggregate framework like *block size* (Section 3.5) and *privacy budget allocation* (Section 3.6). Consider the following motivating example:

Example 2. *Let $T_{70\ yrs}$ and T_{now} be two datasets containing the ages of citizens in a particular region 70 years earlier and at present respectively. It is conceivable that the privacy threat to $T_{70\ yrs}$ is much lower as many of the participating population may have deceased. Although $T_{70\ yrs}$ might not be as useful as T_{now} for learning specific trends about the current population, it can be used to learn some general concepts about T_{now}. For example, a crude estimate of the maximum age present in T_{now} can be obtained from $T_{70\ yrs}$.*

Gupt estimates such general trends in data distribution and uses them to optimize the performance of the system. The optimization results in a significant reduction in error. More precisely, the aged data is used for the following: i) to estimate an optimal block size for use in the sample and aggregate framework, ii) to identify the minimum privacy budget needed to estimate the final result within a given accuracy bound, and iii) to appropriately distribute the privacy budget ϵ across various tasks and queries.

For simplicity of exposition, the particular *aging* model in our analysis is that a constant fraction of the dataset has *completely aged out*, *i.e.* the privacy of the entries in this constant fraction is no more of a concern[2]. In reality, if the aged data is still weakly privacy sensitive, then it is possible to privately estimate these parameters by introducing an appropriate magnitude of noise into these calculations. The weak privacy of aged data allows us to keep the noise low enough such that the estimated parameters are still useful. Existing techniques [6, 140] have attempted to use progressive aggregation of old data in order to reduce its sensitivity. The use of differentially private operations for aggregation can potentially be exploited to generate our training datasets. The use of these complementary approaches offer exciting opportunities that have not been explored in this thesis.

It is important to mention that Gupt *does not* require the *aging* model for default functionality. The default parameter choices allow it work well in a generic setting. However, our experiments show that the *aging* model provides an additional improvement in performance.

3.5 Block size estimation

In this section, we address the following question: *Given a fixed privacy budget, how do we pick the optimal block size to maximize the accuracy of the private output?*

Observe that increasing the block size β increases the noise magnitude, but reduces the estimation error. Therefore, the question boils down to: *how do we select an optimal block size that will allow us to balance the estimation error and the noise?* The following example elucidates why answering the above question is important.

Example 3. *Consider the same age dataset T used in Example 1. If our goal is to find the average of the entries in T while preserving privacy, then it can be observed that (ignoring resampling) the optimal size of each block is one which attains the optimal balance between the estimation error and noise. If the block size was one, then the expected error will be*

[2]This dataset does not have to be aged per-se. Any criteria for selecting a insensitive subset will do

*$O(1/n)$, where n is the size of the dataset. However, if we use the default block size (*i.e., *$n^{0.6}$), the expected error will be $O(1/n^{0.4})$ which is much higher.*

As a result getting the optimal block size based on the specific task helps to reduce the final error to a large extent. The optimal block size varies from problem to problem. For example, in k-means clustering or logistic regression the optimal block size has to be much larger than one.

Let $\ell = n^\alpha$ be the optimal number of blocks, where α is a parameter to be ascertained. Hence, $n^{1-\alpha}$ is the block size. (For the simplicity of exposition we do not consider resampling.) Let $f : \mathbb{R}^{k \times n} \to \mathbb{R}$ be the query which is to be computed on the dataset T. Let the data blocks be represented as $T_1, \cdots, T_\ell$. Let s be the sensitivity of the query, *i.e.,* the absolute value of maximum change that any $f(T_i)$ can have if any one entry of T is changed. With the above parameters in place, the ϵ-differentially private output from the sample and aggregate framework is

$$\hat{f}(T) = \frac{1}{n^\alpha} \sum_{i=1}^{n^\alpha} f(T_i) + \mathsf{Lap}(\frac{s}{\epsilon n^\alpha}) \tag{3.1}$$

Assume that the entries of the dataset T are drawn i.i.d, and that there exists a dataset T^{np} (with entries drawn i.i.d. from the same distribution as T) whose privacy we do not care for under the *aging of sensitivity* model. Let n_{np} be the number of entries in T^{np}. We will use the *aged* dataset T^{np} to estimate the optimal block size. Specifically, we partition T^{np} into blocks of size $\beta = n^{1-\alpha}$. The number of blocks ℓ_{np} in T^{np} is therefore $\ell_{\text{np}} = \frac{n_{\text{np}}}{n^{1-\alpha}}$. Notice that ℓ_{np} is a function of α, whose optimal value has to be found. Also note that the minimum value of α must satisfy the following inequality: $n_{\text{np}} \geq n^{1-\alpha}$.

One possible approach for achieving a good value of α is by minimizing the empirical error in the final output. The empirical error in the output of $\hat{f}$ is defined as

$$\underbrace{\left| \frac{1}{\ell_{np}} \sum_{i=1}^{\ell_{np}} f(T_i^{np}) - f(T^{np}) \right|}_{A} + \underbrace{\frac{\sqrt{2}s}{\epsilon n^\alpha}}_{B} \tag{3.2}$$

Here A characterizes the estimation error, and B is due to the Laplace noise added. We can minimize Equation 3.2 w.r.t. α when $\alpha \in [1 - \log n_{\text{np}} / \log n, 1]$. Conventional techniques like *hill climbing* can be used to obtain a local minima.

The α computed above is used to obtain the optimal number of blocks. Since, the computation involves only the non-private database T^{np}, there is no effect on overall privacy.

3.6 Estimating privacy budget for accuracy goals

In differential privacy, the analyst is expected to specify the privacy goals in terms of an abstract privacy budget ϵ. The analyst performs the data analysis task optimizing it for

accuracy goals and the availability of computational resources. These metrics do not directly map onto the abstract privacy budget. It should be noted that even a privacy expert might be unable to map the privacy budget into accuracy goals for arbitrary problems. In this section we describe mechanisms that Gupt use to convert the accuracy goals into a privacy budget and to efficiently distribute a given privacy budget across different analysis tasks.

In this section we seek to answer the question: *How can Gupt pick an appropriate ϵ, given a fixed accuracy goal?* Specifically, we wish to minimize the ϵ parameter to maximally preserve the privacy budget. It is often more intuitive to specify an accuracy goal rather than a privacy parameter ϵ, since accuracy relates to the problem at hand.

Similar to the previous section, we assume the existence of an *aged* dataset T^{np} (drawn from the same distribution as the original dataset T) whose privacy is not a concern.

Consider an analyst who wishes to guarantee an accuracy ρ with probability $1-\delta$, *i.e.* the output should be within a factor ρ of the true value. We wish to estimate an appropriate ϵ from an aged data set T^{np} of size n_{np}. Let β denote the desired block size. To estimate ϵ, first the permissible standard deviation in the output σ is calculated for a specified accuracy goal ρ and then the following optimization problem is solved. Solve for ϵ, under the following constraints: 1) the expression in Equation 3.3 equals σ^2, 2) $\alpha = \max\{0, \log(n/\beta)\}$.

$$\underbrace{\frac{1}{n^{\alpha}}\left(\frac{1}{\ell_{\text{np}}}\sum_{i=1}^{\ell_{\text{np}}}\left(f(T_i^{\text{np}}) - \frac{1}{\ell_{\text{np}}}\sum_{i=1}^{\ell_{\text{np}}} f(T_i^{\text{np}})\right)^2\right)}_{C} + \underbrace{\frac{2s^2}{\epsilon^2 n^{2\alpha}}}_{D} \tag{3.3}$$

In Equation 3.3, C denotes the variance in the estimation error and D denotes the variance in the output due to noise.

To calculate σ from the accuracy goal, we can rely on Chebyshev's inequality: $\Pr[|\hat{f}(T) - \mathbb{E}(f(T_i))| > \phi\sigma] < \frac{1}{\phi^2}$. Furthermore, assuming that the query f is a approximately normal statistic, we have $|\mathbb{E}(f(T_i)) - \text{Truth}| = O(1/\beta)$. Therefore: $\Pr[|\hat{f}(T) - \text{Truth}| > \phi\sigma + O(1/\beta)] < (1/\phi^2)$ To meet the output accuracy goal of ρ with probability $1-\delta$, we set $\sigma \simeq \sqrt{\delta}|1-\rho|f(T^{\text{np}})$. Here, we have assumed that the *true* answer is $f(T^{\text{np}})$ and $1/\beta \ll \sigma/\sqrt{\delta}$.

Since in the above calculations we assumed that the true answer is $f(T^{\text{np}})$, an obvious question is "why not output $f(T^{\text{np}})$ as the answer?". It can be shown that in a lot of cases, the private output will be much better than $f(T^{\text{np}})$.

If the assumption that $1/\beta \ll \sigma/\sqrt{\delta}$ does not hold, then the above technique for selecting privacy budget would produce suboptimal results. This however does not compromise the privacy properties that Gupt wants to maintain, as it explicitly limit the total privacy budget allocated for queries accessing a particular dataset.

3.7 Distribution of privacy budget between data queries

Differential privacy is an alien concept for most analysts. Further, the proper distribution of the limited privacy budget across multiple computations require significant mathematical expertise. Gupt eliminates the need to manually distribute privacy budget between tasks. The following example will highlight the requirement of an efficient privacy budget distribution rather than distributing equally among various tasks.

Example 4. *Consider the same age census dataset T from Example 1. Suppose we want to find the average age and the variance present in the dataset while preserving differential privacy. Assume that the maximum possible human age is max and the minimum age is zero. Assume that the non-private variance is computed as $\frac{1}{n}\sum_{i=1}^{n}(T(i) - Av_{priv})^2$, where Av_{priv} is the private estimate of the average and n is the size of T. If an entry of T is modified, the average Av changes by at most max/n, however the variance can change by at most max^2/n.*

Let ϵ_1 and ϵ_2 be the privacy level expected for average and variance respectively, with the total privacy budget being $\epsilon = \epsilon_1 + \epsilon_2$. Now, if it is assumed that $\epsilon_1 = \epsilon_2$, then the error in the computation of variance will be in the order of max more than in the computation of average. Whereas if privacy budget were distributed as $\epsilon_1 : \epsilon_2 = 1 : max$, then the noise in both the average and variance will roughly be the same.

Given privacy budget of ϵ and we need to use it for computing various queries $f_1, \cdots, f_m$ privately. If the private estimation of query f_i requires ϵ_i privacy budget, then the total privacy budget spent will be $\sum_{i=1}^{m}\epsilon_i$ (by *composition* property of differential privacy [31]). The privacy budget is distributed as follows. Let $\frac{\zeta_i}{\epsilon_i}$ be the standard deviation of the Laplace noise added by Gupt to ensure privacy level ϵ_i. Allocate the privacy budget by setting $\epsilon_i = \frac{\zeta_i}{\sum_{i=1}^{m}\zeta_i}\epsilon$. The rationale behind taking such an approach is that usually the variance in the computation by Gupt is mostly due to the variance in the Laplace noise added. Hence, distributing ϵ across various tasks using the technique discussed above ensures that the variance due to Laplace noise in the private output for each f_i is the same.

3.8 Theoretical guarantees for privacy and utility

Gupt guarantees ϵ-differential privacy to the final output. It provides similar utility guarantees as the original sample and aggregate algorithm from [130]. This guarantee applies to the queries satisfying "approximate normality[3]" condition defined by Smith [130], who also observed that a wide class of queries satisfy this normality condition. Some of the examples being various maximum-likelihood estimators and estimators for regression problems.

[3]By approximately normal statistic we refer to the *generic* asymptotically normal statistic in Definition 2 of [130].

Gupt provides the same level of privacy for queries that are not approximately normal. Reasonable utility could be expected even from queries that are not approximately normal, even though no theoretical guarantees are provided.

We combine the privacy guarantees of the different pieces used in the system to present a final privacy guarantee. We provide three different privacy guarantees based on how the output range is being estimated.

Theorem 1 (Privacy Guarantee for GUPT)**.** *Let* $T \in \mathbb{R}^{k \times n}$ *be a dataset and* $f : \mathbb{R}^{k \times n} \to \mathbb{R}^p$ *be the query. Gupt is* ϵ*-differentially private if the following holds:*

1. **Gupt uses `GUPT-helper` with loose range for the input:** *Execute percentile estimation algorithm defined in [130] for each of the* k *input dimensions with privacy parameter* $\epsilon/(2k)$ *and then run the sample and aggregate framework (SAF) with privacy parameter* $\epsilon/(2p)$ *for each output dimension.*

2. **Gupt uses `GUPT-tight`:** *Run SAF with privacy parameter* ϵ/p *for each output dimension.*

3. **Gupt uses no bounds on the input data:** *Execute percentile estimation algorithm defined in [30] twice for each of the* k*-dimensions of the dataset (once for the* 1^{st} *quartile and once for the* 3^{rd}*) with privacy parameter* $\frac{\epsilon}{18k}$*. Run the sample and aggregate framework with privacy parameter* $\frac{\epsilon}{2p}$ *for each of the output dimensions. The* δ *parameter in this case is* $negl(n)$*, where* n *is the size of the dataset.*

4. **Gupt uses `GUPT-loose`:** *For each output dimension, run the percentile estimation algorithm defined in [130] with privacy parameter* $\epsilon/(2p)$ *and then run SAF with privacy parameter* $\epsilon/(2p)$*.*

The proof of this theorem directly follows from the privacy guarantees for each of the module of Gupt and the composition theorem of [31]. In terms of utility, we claim the following about Gupt.

Theorem 2 (Utility Guarantee for Gupt)**.** *Let* $f : \mathbb{R}^{k \times n} \to \mathbb{R}^p$ *be a generic asymptotically normal statistic (see Definition 2 in [130]). Let* T *be a dataset of size* n *drawn i.i.d. from some underlying distribution* F*. Let* $\hat{f}(T)$ *be the statistic computed by Gupt.*

1. *If Gupt uses* `GUPT-tight`*, then* $\hat{f}(T)$ *converges to* $f(T)$ *in distribution.*

2. *Let* f *be a statistic which differ by at most* γ *(under some distance metric* d*) on two datasets* T *and* $\tilde{T}$*, where* $\tilde{T}$ *is obtained by clamping the dataset* T *on each dimension by the* 75*-th percentile and the* 25*-th percentile for that dimension. If Gupt uses* `GUPT-helper` *with loose input range, then we have* $d(\hat{f}(\tilde{T}), f(T)) \leq \gamma$ *as* $n \to \infty$*.*

3. *If Gupt uses* `GUPT-loose`*, then* $\hat{f}(T)$ *converges in distribution to* $f(T)$ *as* $n \to \infty$ *as long as* k, $\frac{1}{\epsilon}$ *and* $\log(|\max - \min|)$ *are bounded from above by sufficiently small polynomials in* n*, where* $|\max - \min|$ *is the loose output range provided.*

The proof follows using a similar analysis used in [130].

3.9 System security

Gupt is designed as a hosted platform where the analyst is *not* trusted. It is thus important to ensure that the untrusted computation should not be able to access the datasets directly. Additionally, it is important to prevent the computation from exhausting resources or compromising the service provider. To this end, the "computation manager" is split into a server component that interacts with the user and a client component that runs on each node in the cluster. The trusted client is responsible for instantiating the computation in an isolated execution environment. The isolated environment ensures that the computation can only communicate with a trusted forwarding agent which sends the messages to the computation manager.

3.9.1 Access control

Gupt uses a mandatory access control framework (MAC) to ensure that (a) communication between different instances of the computation is disallowed and (b) each instance of the computation can only store state (or modify data) within its own scratch space. This is the only component of Gupt that depends upon a platform dependent implementation. On Linux, the LSM framework [142] has enabled many MAC frameworks such as SELinux and AppArmor to be built. Gupt defines a simple AppArmor policy for each instance of the computation, setting its working directory to a temporary scratch space that is emptied upon program termination. AppArmor does not yet allow fine grained control to limit network activity to individual hosts and ports. Thus the "computation manager" is split into a server and client component. The client component of the computation manager allows Gupt to disable all network activity for the untrusted computation and restrict IPC to the client.

We determined an empirical estimate of the overhead introduced by the AppArmor sandbox by executing an implementation of k-means clustering on Gupt $6,000$ times. We found that *the sandboxed version of Gupt had an overhead of* 1.26% *over the non-sandboxed version).*

3.9.2 Protection against side-channel attacks

Haeberlen *et al.* [51] identified three possible side-channel attacks against differentially private systems. They are i) *state attack*, ii) *privacy budget attack*, and iii) *timing attack*. Gupt is not vulnerable to any of these attacks.

State attacks: If the adversarial program can modify some internal state (*e.g.,* change the value of a static variable) when encountered with a specific data record. An adversary can then look at the state to figure out whether the record was present in the dataset. Both PINQ (in it's current implementation) and Airavat are vulnerable to state attacks. However, it is conceivable that operations can be isolated using .NET *AppDomain*s in PINQ to isolate data computations. Since Gupt executes the complete analysis program (which may be adversarial) in isolated execution chambers and allows the analyst to access only the final differentially private output, state attacks are automatically protected against.

Privacy budget attack: In this attack, on encountering a particular record, the adversarial program issues additional queries that exhausts the remaining privacy budget. [51] noted that PINQ is vulnerable to this attack. Gupt protects against privacy budget attacks by managing the privacy budget itself, instead of letting the untrusted program perform the budget management.

Timing attacks: In a timing attack, the adversarial program could consume an unreasonably long amount of time to execute (perhaps get into an infinite loop) when encountered with a specific data record. Gupt protects against this attack by setting a predefined bound on the number of cycles for which the data analyst program runs on each data block. If the computation on a particular data block completes before the predefined number of cycles, then Gupt waits for the remaining cycles before producing an output from that block. In case the computation exceeds the predefined number of cycles, the computation is killed and a constant value within the expected output range is produced as the output of the program running on the data block under consideration.

Note that with the scheme above, the runtime of Gupt is independent of the data. Hence, the number of execution cycles does not reveal any information about the dataset. The proof that the final output is still differentially private under this scheme follows directly from the privacy guarantee of the sample and aggregate framework and the fact that a change in one data entry can affect only one data block (ignoring resampling). Thus Gupt is not vulnerable to timing attacks. Both PINQ and Airavat do not protect against timing attacks [51].

3.10 Evaluation of parameter sensitivity in Gupt

For each data analysis program, the program binary and interfaces with the Gupt "computation manager" should be provided. For arbitrary binaries, a lean wrapper program can be used for marshaling data to/from the format of the computation manager.

In this section, we show using results from running common machine learning algorithms (such as k-means clustering and logistic regression on a life sciences dataset) that Gupt does not significantly affect the accuracy of data analysis. Further, we show that Gupt not only relieves the analysts from the burden of distributing a privacy budget between data transformation operations, it also manages to provide superior output accuracy. Finally, we show through benchmarks the scalability of the Gupt architecture and the benefits of using aged data to estimate optimal values of privacy budget and block sizes.

We evaluate the efficacy of Gupt using the `ds1.10` life sciences dataset taken from `http://komarix.org/ac/ds` as a motivating example for data analysis. This dataset contains the top 10 principal components of chemical/biological compounds with each of the 26,733 rows representing different compounds. Additionally, the reactivity of the compound is available as an additional component. A k-means clustering experiment enables us to cluster compounds with similar features together and logistic regression builds a linear classifier for the experiment (*e.g.,* predicting carcinogens). It should be noted that these experiments only provide estimates as the final answer, *e.g.,* the cluster centroids in the case of k-means. We show in this section that the perturbation introduced by Gupt only affects the final result marginally.

3.10.1 Privacy budget distribution

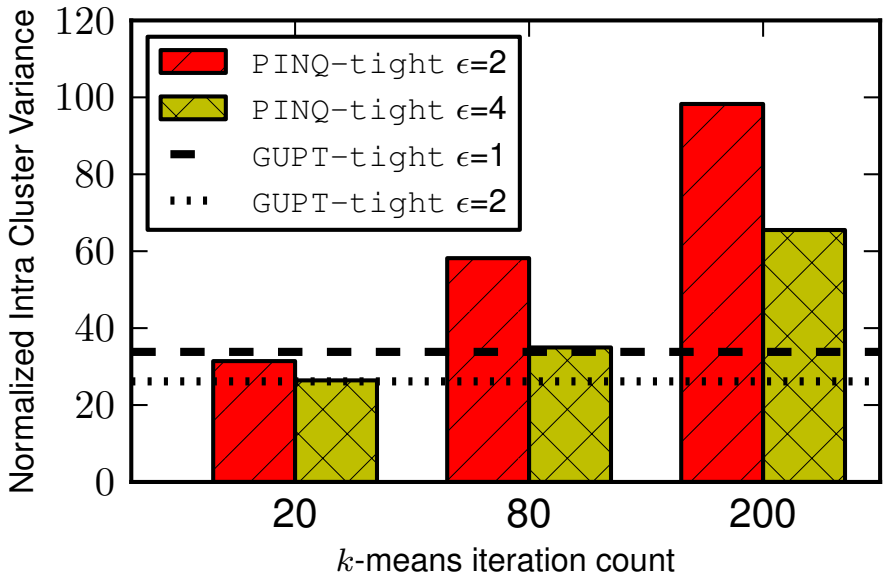


Figure 3.2: Total perturbation introduced by Gupt does not change with number of operations in the utility function

In Gupt, the program is treated as a black box and noise is only added to the output of the entire program. Thus the number of operations performed in the program itself is irrelevant. A problem with writing specialized differentially private algorithms such as in the case of PINQ is that given a privacy budget ϵ for the task, it is difficult to decide how much ϵ to spend on each query, since it is difficult to determine the number of iterations needed ahead of time. PINQ requires the analyst to pre-specify the number of iterations in order to allocate the privacy budget between iterations. This is often hard to do, since many

data analysis algorithms such as PageRank [108] and recursive relation queries [13] require iterative computation until the algorithm reaches convergence. The performance of PINQ thus depends on the ability to accurately predict the number of iterations. If the specified number of iterations is too small, then the algorithm may not converge. On the other hand, if the specified number of iterations is too large, then much more noise than is required will be added which will both slow down the convergence of the algorithm as well as harm its accuracy. Figure 3.2 shows the effect of PINQ on accuracy when performing k-means clustering on the dataset. In this example, the program output for the dataset converges within a small number of iterations, e.g., $n = 20$. Whereas if a larger number of iterations (e.g., $n = 200$) was conservatively chosen, then PINQ's performance degrades significantly. On the other hand, Gupt produces the same amount of perturbation irrespective of the number of iterations in k-means. Further, it should be noted that PINQ was subjected to a weaker privacy constraint ($\epsilon = 2$ and 4) as compared to Gupt ($\epsilon = 1$ and 2).

3.10.2 Privacy budget estimation

GUPT uses an aged dataset (that is no longer considered privacy sensitive) drawn from a similar distribution as the real dataset. Section 3.5 above describes the use of aged data to estimate an optimal block size that reduces the error introduced by data sampling. Section 3.6 describes how data analysts who are not privacy experts can continue to only describe their accuracy goals yet achieve differentially private outputs. Finally, Section 3.7 uses aged data to automatically distribute a privacy budget between different queries on the same data set. In this section, we show experimental results that support the claims made in Sections 3.5 and 3.6.

To illustrate the ease with which Gupt can be used by data analysts, we evaluate the efficiency of Gupt by executing queries that are not provided with a privacy budget. We use a census income dataset from the UCI machine learning repository [40] which consists of 32561 entries. The age data from this dataset is used to calculate the average age. A reasonably loose range of $[0, 150]$ was enforced on the output whose true average age is 38.5816. Initially, the experiment was run with a constant privacy budgets of $\epsilon = 1$ and $\epsilon = 0.3$. Gupt allows the analyst to provide looser constraints such as "90% result accuracy for 90% of the results" and allocates only as much privacy budget as is required to meet these properties. In this experiment, the 10% of the dataset was assumed to be completely privacy insensitive and was used to estimate ϵ given a pre-determined block size. Figure 3.3 shows the CDF of the output accuracy both for constant privacy budget values as well as for the accuracy requirement. Interestingly, not only does the figure show that the accuracy guarantees are met by GUPT, but also it shows that if the analyst was to define the privacy budget manually (as in the case of $\epsilon = 1$ or $\epsilon = 0.3$), then either too much or too little privacy budget is used. The privacy budget estimation technique thus has the additional advantage

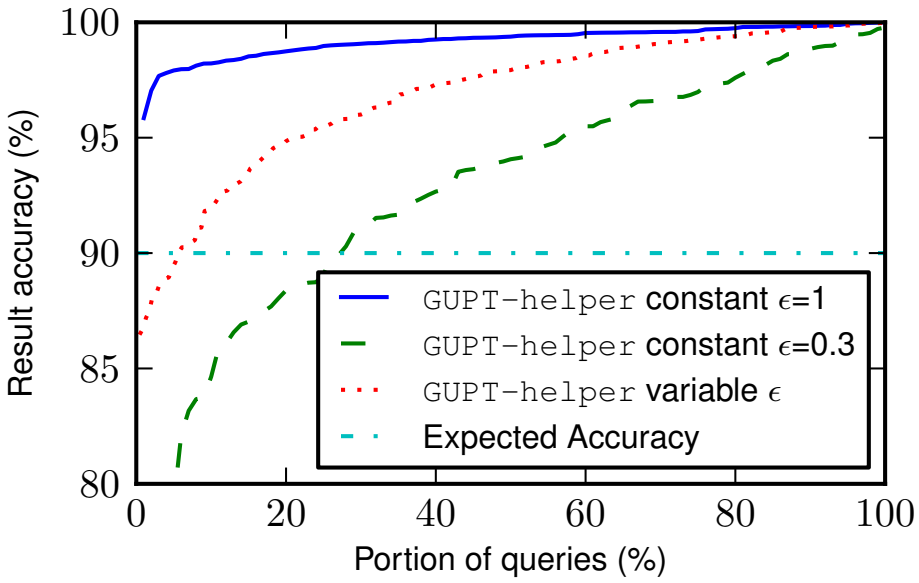


Figure 3.3: CDF of query accuracy for privacy budget allocation mechanisms

that the lifetime of the total privacy budget for a dataset will be extended. Figure 3.4 shows that if we were to run the average age query with the above constraints over and over again, Gupt will be able to run 2.3 times more queries than using a constant privacy budget of $\epsilon = 1$.

3.10.3 Block size estimation

Section 3.5 shows that the estimation error decreases with an increase in data block size, whereas the noise decreases with an increased number of blocks. The optimal trade off point between the block size and number of data blocks would be different for different queries executed on the dataset. To illustrate the trade-off, we show results from queries executed on an internet advertisement dataset also from the UCI machine learning repository [40]. Figure 3.5 shows the normalized root mean square error (from the true value) in estimating the mean and median aspect ratio of advertisements shown on Internet pages with privacy budgets ϵ of 2 and 6. In the case of the "mean" query, since the averaging operation is already performed by the sample and aggregate framework, smaller data blocks would reduce the noise added to the output and thus provide more accurate results. As expected, we see that the ideal block size would be one.

For the "median" query, it is expected that increasing the block size would generate more accurate inputs to the averaging function. Figure 3.5 shows that when the "median" query is executed with $\epsilon = 2$, the error is minimal for a block size of 10. With increasing block sizes, the noise added to compensate for the reduction in number of blocks would have

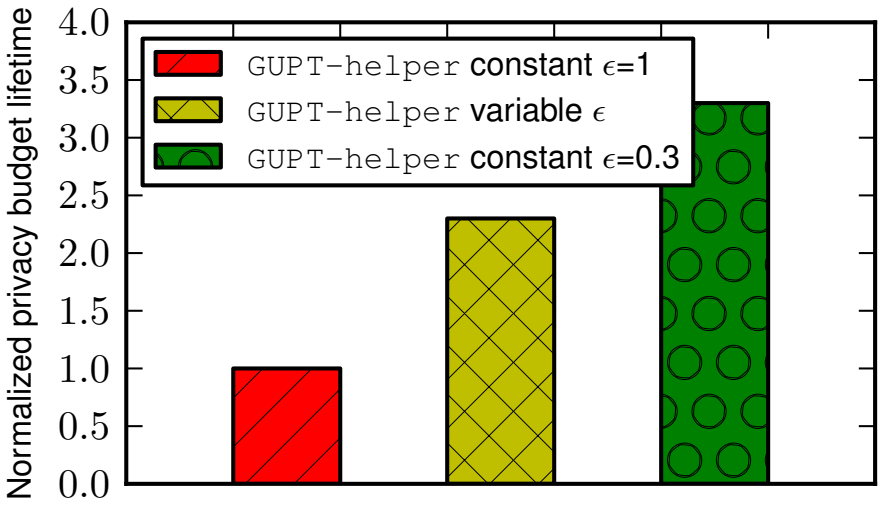


Figure 3.4: Increased lifetime of total privacy budget using privacy budget allocation mechanism

a dominating effect. On the other hand, when executing the same query with $\epsilon = 6$, the error continues to drop for increased block sizes, as the estimation error dominates the Laplace noise (owing to the increased privacy budget). It is thus clear that Gupt can significantly reduce the total error by estimating the optimal block size for the sample and aggregate framework.

3.10.4 Accuracy of output

As mentioned in Section 3.3, any analysis performed using Gupt has two sources of error – (a) an estimation error, introduced because each instance of the computation works on a smaller subset of the data and (b) Laplace noise that is added in order to guarantee differential privacy. In this section, we show the effect of these errors when running logistic regression and k-means on the life sciences dataset.

GUPT can be used to run existing programs with no modifications, thus drastically reducing the overhead of writing privacy preserving programs. Analysts using Gupt are free to use their favorite software packages written in any language. To demonstrate this property, we evaluate black box implementations of logistic regression and k-means clustering on the life sciences dataset.

Logistic Regression: The logistic regression software package from Microsoft Research

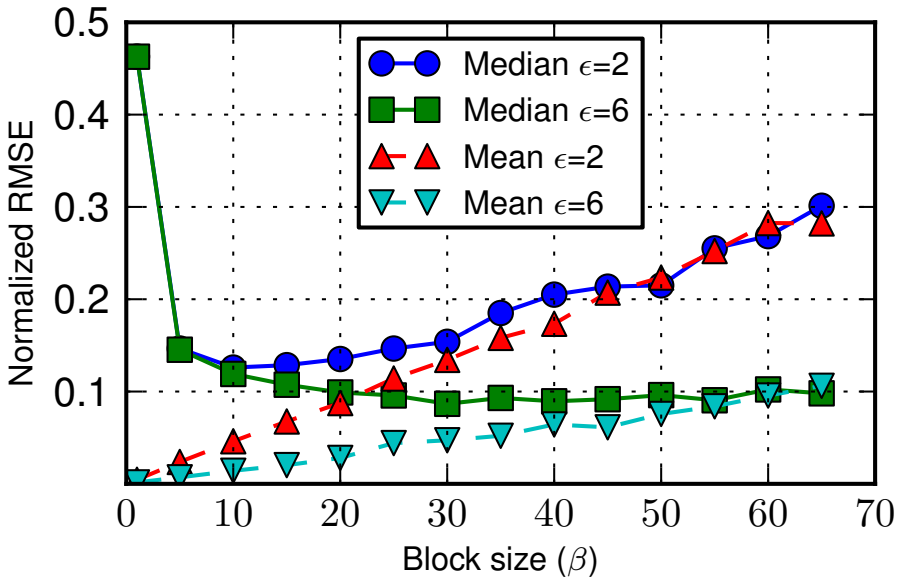


Figure 3.5: Change in error for different block sizes

(*Orthant-Wise Limited-memory Quasi-Newton Optimizer for L_1-regularized Objectives*) was used to classify the compounds in the dataset as carcinogens and non-carcinogens. Figure 3.6 shows the accuracy of Gupt for different privacy budgets.

When the package was run on the dataset directly, a baseline accuracy of 94% was obtained. The same package when run using the Gupt framework classified carcinogens with an accuracy between $75 \sim 80\%$. To understand the source of the error, when the non-private algorithm was executed on a data block of size $\frac{n}{n^{0.4}}$ records, the accuracy reduced to 82%. It was thus determined that much of the error stems from the loss of accuracy when the algorithm is run on smaller blocks of the entire dataset reduced. For datasets of increasingly large size, this error is expected to diminish.

k-means Clustering: Figure 3.7 shows the cluster variance computed from a k-means implementation run on the life sciences dataset. The x-axis is various choices of the privacy budget ϵ, and the y-axis is the normalized Intra-Cluster Variance (ICV) defined as $\frac{1}{n}\sum_{i=1}^{K}\sum_{\vec{x}\in C_i}|\vec{x}-\vec{c}_i|_2^2$, where K denotes the number of clusters, C_i denotes the set of points within the i^{th} cluster, and $\vec{c}_i$ denotes the center of the i^{th} cluster. A standard k-means implementation from the `scipy` python package is used for the experiment.

The k-means implementation was run using Gupt with different configurations for calculating the output range (Section 3.3.1). For `GUPT-tight`, a tight range for the output is taken to be the exact minimum and the maximum of each attribute (for all 10 attributes). For `GUPT-loose`, a loose output range is fixed as $[\min * 2, \max * 2]$, where min and max are the actual minimum and maximum for that attribute. Figure 3.7 shows that with increasing privacy budget ϵ, the amount of Laplace noise added to guarantee differential privacy de-

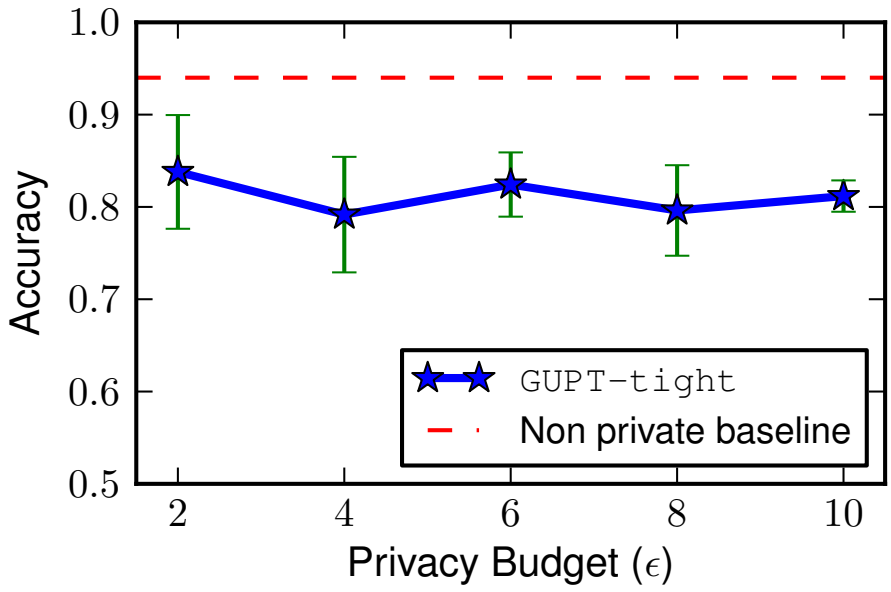


Figure 3.6: Effect of privacy budget on the accuracy of prediction using Logistic Regression on the life sciences dataset

creases, thereby reducing the intra-cluster variance, i.e., making the answer more accurate. It can also be seen that when Gupt is provided with reasonably tight bounds on the output range (`GUPT-tight`), the output of the k-means experiment is very close to a non-private run of the experiment even for small values of the privacy budget. If only loose bounds are available (`GUPT-loose`), then a larger privacy budget is required for the same output accuracy.

3.10.5 Scalability

Using a server with two Intel Xeon 5550 quad-core CPUs and the entire dataset loaded in memory, we compare the execution time of an unmodified (non-private) instance and a Gupt instance of the k-means experiment.

If tight output range (i.e.,, `GUPT-tight`) is not available, typically, the output range estimation phase of the sample and aggregate framework takes up most of the CPU cycles. When only loose range for the input is available (i.e.,, `GUPT-helper`), a differentially private percentile estimation is performed on all of the input data. This is a $O(n \ln n)$ operation, n being the number of data records in the original dataset. On the other hand, if even loose range for the output is available (i.e.,, `GUPT-loose`), then the percentile estimation is performed only on the output of each of the blocks in sample and aggregate framework, which is typically around $n^{0.4}$. This results in significantly reduced run-time overhead. The overhead introduced by Gupt is irrespective of the actual computation time itself. Thus as

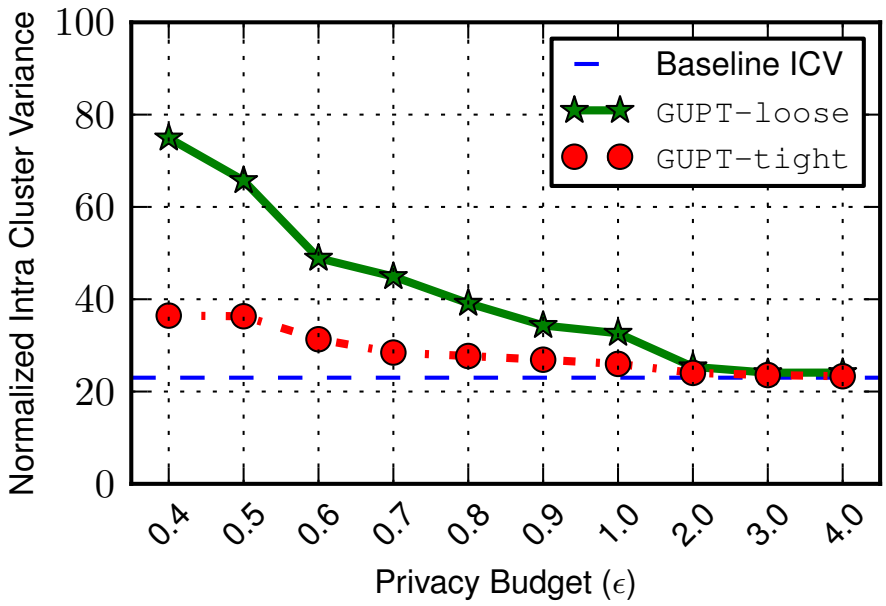


Figure 3.7: Intra-cluster variance for k-means clustering on the life sciences dataset

the computation time increases, the overhead introduced by Gupt diminishes in comparison. Further, there is an additional speed up since each of the computation instances work on a smaller subset of the entire dataset. It should be noted that the reduction in computation time thus achieved could also potentially be achieved by the computational task running without GUPT. Figure 3.8 shows that the overall completion time of the private versions of the program increases slowly compared to the non-private version as we increase the number of iterations of k-means clustering.

3.11 Qualitative comparison with other differential privacy platforms

In this section, Gupt is contrasted with both PINQ and Airavat on various fronts (see Table 3.1 for a summary). We also list the significant changes introduced by Gupt in order to mold the sample and aggregate framework (SAF) [130] into a practically useful one.

Unmodified programs: Because PINQ [90] is an API that provides a set of low-level data manipulation primitives, applications will need to be re-written to perform all operations using these primitives. On the other hand, Airavat [119] implements the Map-Reduce programming paradigm [26] and requires that the analyst splits the user's data analysis program into an "untrusted" map program and a reduce aggregator that is "trusted" to be differentially private.

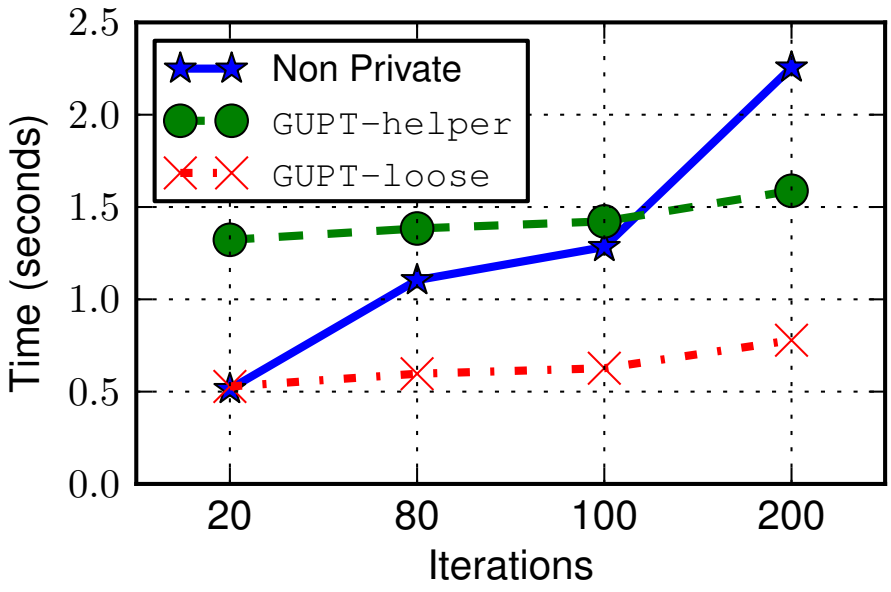


Figure 3.8: Change in computation time for increased number of iterations in k-means

In contrast, Gupt treats the complete application program as a black box and as a result the entire application program is deemed untrusted.

Expressiveness of the program: PINQ provides a limited set of primitives for data operations. However, if the required primitives are not already available, then a privacy unaware analyst would be unable to ensure privacy for the output. Airavat also severely restricts the expressiveness of the programs that can run in it's framework: a) the "untrusted" map program is completely isolated for each data element and cannot save any global state and b) it restricts the number of key-value pairs generated from the mapper. Many machine learning algorithms (such as clustering and classification) require global state and complex aggregation functions. This would be infeasible in Airavat without placing much of the logic in the "trusted" reducer program.

GUPT places no restriction on the application program, and thus does not degrade the expressiveness of the program.

Privacy budget distribution: As was shown in Section 3.10.1, PINQ requires the analyst to allocate a privacy budget for each operation on the data. An inefficient distribution of the budget either add too much noise or use up too much of the budget. Like Gupt, Airavat spends a constant privacy budget on the entire program. However, neither Airavat nor PINQ provides any support for distributing an aggregate privacy budget across multiple data analysis programs.

Using the *aging of sensitivity* model, Gupt provides an mechanism for efficiently distributing an aggregate privacy budget across multiple data analysis programs.

Side Channel Attacks: Current implementation of PINQ lays the onus of protecting

	Gupt	**PINQ**	**Airavat**
Works with unmodified programs	Yes	No	No
Allows expressive programs	Yes	Yes	No
Automated privacy budget allocation	Yes	No	No
Protection against privacy budget attack	Yes	No	Yes
Protection against state attack	Yes	No	No
Protection against timing attack	Yes	No	No

Table 3.1: Comparison of GUPT, PINQ and Airavat

against side channel attacks on the program developer. As was noted in [51], although Airavat protects against privacy budget attacks, it remains vulnerable to state attacks. Gupt defends against both state attacks and privacy budget attacks (see Section 3.9.2).

Other differences: GUPT extends SAF to use a novel data resampling mechanism to reduce the variance in the output induced via data sub-sampling. Using the *aging of sensitivity* model, Gupt overcomes a fundamental issue in differential privacy not considered previously (to the best of our knowledge): *for an arbitrary data analysis application, how do we describe an abstract privacy budget in terms of utility?* The model also allows us to further reduce the error in SAF by estimating a reasonable block size.

3.12 Summary

GUPT introduces a new model for data sensitivity which applies to a large class of datasets where the privacy requirement of data decreases over time. As we will explain in Section 3.4, using this model is appropriate and allows us to overcome significant challenges that are fundamental to differential privacy. This approach enables us to analyze less sensitive data to get reasonable approximations of privacy parameters that can be used for data queries running on the newer or more sensitive data. GUPT makes the following technical contributions that make differential privacy usable in practice:

1. **Describing privacy budget in terms of accuracy:** Data analysts are accustomed to the idea of working with inaccurate output (as is the case with data sampling in large datasets and machine learning algorithms have probabilistic output). Gupt uses the aging model of data sensitivity to allow analysts to describe the abstract 'privacy budget' in terms of expected accuracy of the final output.

2. **Privacy budget distribution:** Gupt automatically allocates a privacy budget to each query in order to match the data analysts' overall accuracy requirements. Further, the analyst also does not have to explicitly distribute the privacy budget between the individual data operations in the program.

3. **Accuracy of output:** Gupt extends a theoretical differential privacy framework called "sample and aggregate" (described in Section 2.3) for practical applicability. This includes using a novel data resampling technique that reduces the error introduced by the framework's data partitioning scheme. Further, the aging model of data sensitivity allows Gupt to select an optimal partition size that reduces the perturbation added for differential privacy.

4. **Prevent side channel attacks:** Gupt defends against side channel attacks such as the privacy budget attacks, state attacks and timing attacks described in [51].

Chapter 4

Enforcing user privacy policies

Enterprises hosting applications in cloud-based data centers enjoy increased flexibility in resource allocation and lower upfront capital costs. However, the migration of applications from on-premises data centers to cloud-based hosting also moves the control of sensitive data from the user to the application developer. A user today has to trust that cloud application developers will make judicious decisions to protect the user's privacy, including hardening their operating systems, promptly applying security updates, and using appropriate authentication, encryption and information flow control mechanisms correctly. However, benign developers regularly skip best practices for secure application development—in fact, security misconfigurations have been rated as the 6th most dangerous web application vulnerability due to their prevalence and risk to organizations [106]. Worse, a malicious developer can deliberately misuse her overarching access to users' data [133] or circumvent existing permission-based mechanisms [24] to compromise users' privacy. Users are thus unwilling to use applications from unknown developers in spite of many advertised features. Ideally, the task of specifying and enforcing privacy rules should be separated from individual applications and developers and done *once* and for *all* applications by a trusted underlying system.

4.1 Specifying security policies using ACLs

Access Control Lists (ACLs) are widely used to enforce privacy policies and restrict access to data. Users are already familiar using this mechanism in cloud applications (52% of the top 100 most popular cloud applications were found to make use of ACLs). Specifying privacy policies in the form of Access Control Lists on folders allows users to specify policies on *data* instead of on *applications*. ACLs on folders not only allows a user to control which other users can access the data, but also matches the expectation that applications and cloud services exist simply to provide functionality and thus cannot change data sharing rules [116]. However, once a malicious (or buggy) application has received access to private data, it can leak the information to external entities.

Using this simple access control mechanism as a privacy policy model fails when an untrusted application actually needs to execute on private data. For instance, an untrusted text editor might mix data from a sensitive document into a publicly accessible document without the user's consent as long as both documents are accessible by the user. Restricting this data-mixing requires information flow control techniques that allow users to create security labels and enforce rules on the flow of these labels [28]. Information flow control techniques, however, have a major Achilles heel: the onus is on the application developer to partition the application into discrete components that together achieves its overall functionality. The user now has to either trust the developer to create security labels and partition the program appropriately or have the programs be terminated with security exceptions [150].

Rubicon is a system that brings together the advantages of both approaches (access control lists and information flow control) to enforce user-defined policies without sacrificing security and functionality. End-users express their privacy policies on *data capsules*—a bag of arbitrary bits that the user groups together. *Rubicon allows privacy policies to be expressed by users as simple access control lists (ACLs) instead of working with intricate computer security concepts.* In order to translate ACLs into information flow rules, Rubicon proposes simple extensions to the 3-tier system design pattern (which separates the system into a presentation tier, application tier and data tier) and defines communication channels between the different components. Because a large number of existing systems are already designed using the multi-tiered architecture [113], Rubicon is readily usable for these applications.

This modified design pattern, which we call the Application-Storage-Template pattern (AST) uses a form of robust declassification [100] allowing users to specify privacy policies as ACLs and obtain the privacy guarantees provided by information flow control. The AST design pattern enables developers to implement both user-facing functionality and backend optimizations. In Rubicon, applications perform computations on data capsules within isolated containers and thus cannot leak information to unintended recipients. *AST enables rich applications to be written without requiring security expertise from the developers by effectively mapping simple access control rules to information flow policies.*

Privacy guarantees in Rubicon are defined with reference to **data capsules**. Data capsules are similar to folders in cloud-based file systems such as Google Drive or Dropbox, where a user can create a folder and a user who owns a folder can share it explicitly with others who are signed up with Google or Dropbox.

Every data capsule has a key k (i.e., uniqueID of the folder) and a value $data[k]$, which is an abstraction of the content of the folder that can contain many different files sharing the same policies. The ACL for capsule k consists of users (subscribers of the service) that own, can read and can write $data[k]$, denoted with $owner[k], reader[k]$ and $writer[k]$ respectively. By default a data capsule k is owned by a single user, who belongs both in $reader[k]$ and in $writer[k]$.

Rubicon provides a user with the security invariant that her ACLs are enforced and *an untrusted AST application that executes on Rubicon can never leak data to an unauthorized user.* For example if Alice has chosen not to share her financial expenses document with Bob, Bob will not be able to access this file even when Alice processes the file with some untrusted,

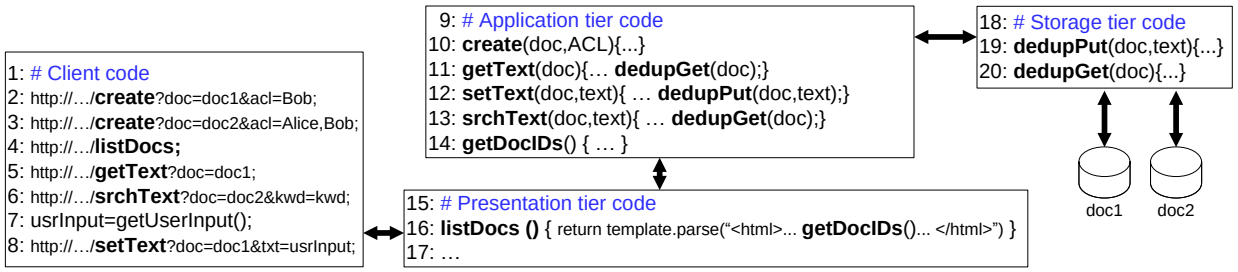


Figure 4.1: An example 3-tiered text editor application. If the client components were left as is, OS-level information flow control enforcement would be unable to let it run without exception, since mixing information from documents doc1 and doc2 with different ACLs would cause overtainting (as the respective calls getText() and searchText() are server by the same application instance).

third-party AST application that executes on Rubicon. In effect, users can control access to a capsule's *content* while Rubicon translates ACLs into information flow restrictions on untrusted applications. Section 4.5 provides a formal design description that shows how Rubicon's ensures that data from capsule k reaches user u iff $u \in reader[k]$, and that data from user u reaches capsule k iff $u \in writer[k]$.

4.2 Motivating example

The example code in Figure 4.1 is used to outline the ease of executing existing 3-tier applications on the RUBICON platform. In the process we highlight various limitations in existing approaches, where functionality has to be sacrificed to achieve security. The program in Figure 4.1 is a proxy for a web-based, text-editing application (e.g., Google docs) that has a client, a cloud and a storage component.

The application exposes a REST API that allows clients to interface with the cloud component. Documents can be created with specific ACLs using the create() interface. Client can access existing documents with the setText(doc,text) and getText(doc) interfaces. In order to optimize storage, the cloud component of the application uses a storage tier that implements *deduplication* [94] (i.e., if two users upload the same document, the deduplication mechanism will only need to store a *single instance* of it). The deduplication component can be accessed through a simple dedupPut/dedupGet interface, again using a REST-based API. Every document has a different security label (e.g., document doc2 has the label "only Bob and Alice can read and write"), that is derived by the ACL at the time of the document's creation (lines 2-3).

The program in Figure 4.1 works with two documents doc1 and doc2. In line 4, a user

first lists the documents that the user owns in the application—doc1 and doc2. The user then reads the text of doc1 and performs a search on doc2 (line 6). Finally, the user opens document doc1 for editing (line 8) through a remote call to setText(). Specifically, to edit this document, the program receives new content as the external input usrInput. When setText() is triggered, the cloud side component stores the new content via the call dedupPut().

The objective of an underlying security mechanism is to ensure that the labels of different documents are not mixed when the program executes, even if the application code (e.g., setText()) is untrusted. This means that after the execution of the program, every document should be tainted only with its own label. We clarify the limitations of existing security mechanisms by analyzing how two common information flow control techniques will enforce this property for the program in Figure 4.1.

OS-based techniques cause run-time exceptions: OS-based Information Flow Control (IFC) techniques [33,71,151] can be used to secure the program of Figure 4.1. Such techniques can cause run-time security exceptions due to conservative labeling by the underlying system: For example, in an OS-based IFC system, the program in Figure 4.1 will be assigned the most conservative label once it reads multiple documents (by line 6). In information flow terms, the label attached to any data written to by the program will be the union of all the documents' confidentiality and integrity labels. When the process tries to write to document doc1 at line 8, it will be terminated at run-time even though the program's logic does not mix data from multiple documents—variable usrInput is an external input and no data from doc2 (line 6) is used in line 8. In this way, a benign piece of code, would be treated as a piece of code whose logic indeed mixes data of different labels, e.g., one that counts the occurrences of kwd across documents and writes the count into usrInput.

A run-time exception could be thrown even if lines 5-8 accesses *only one document*, thus not mixing security labels (assume that creating the documents in lines 2-3 is done from different clients which have different address spaces, therefore labels are not mixed in lines 2-3). This is because simply calling setText() at line 8 would again give the most conservative label to the program, for setText() calls dedupPut(), which in turn requires access to *all documents*, in order to implement the deduplication logic. A similar problem could exist if the storage library was implementing other storage optimizations, such as building compression dictionaries across multiple files [83].

This form of conservative labeling significantly limits the application functionality of programs such as text editors and document readers. Applications that process multiple categories of files and use storage services that require access to a monolithic database (e.g., deduplication) will be terminated as being insecure (even if their code is benign).

Language-based techniques require security expertise: To avoid the above problems of limited functionality and run-time exceptions (so that benign code can still execute), language-based methods could be used. Language-based methods require application variables to be annotated with security labels [120]. For example, in the program of Figure 4.1, variable usrInput would have no taint in its security label, since it is an external input and no information is transferred from documents doc1 and doc2 to this variable—therefore the

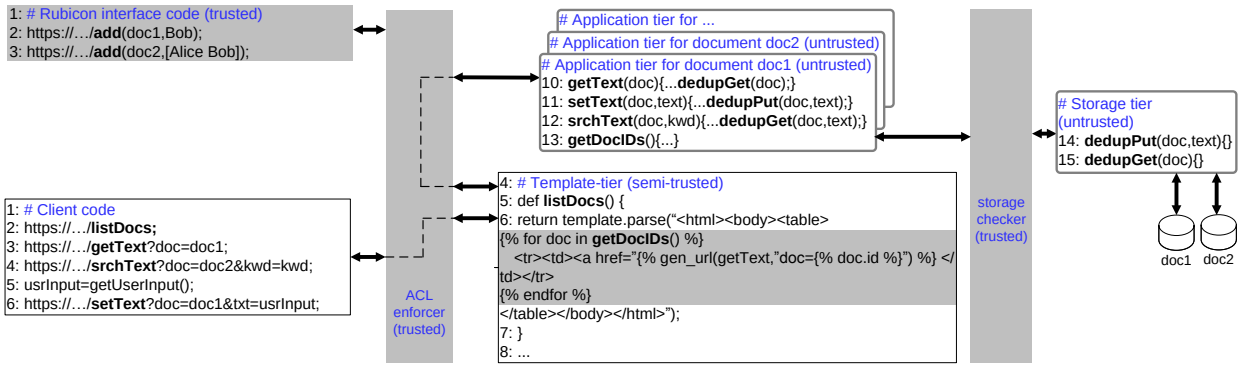


Figure 4.2: The Rubicon translated architecture for the code in Figure 4.1 using our AST pattern. Notice that the presentation, application and storage tier map to AST readily with very few modifications. The trusted components in Rubicon are colored with gray. Untrusted code executes in isolation. The important things to notice is that (a) the application tier only talks to the storage tier through the storage checker; (b) the presentation tier only talks to the application tier through the ACL enforcer.

program would not be terminated at line 8. However, labeling variables requires the developer both to be trusted and to have significant security expertise. In our model, developers are untrusted and thus could maliciously label the code. Moreover, even if they are not malicious they may not have the requisite security expertise to work with complicated information flow labels. Finally, we note here that language-based approaches could also rely on run-time instrumentation ([4, 23, 103, 132]), which however miss implicit information flows and incur heavy performance overheads[1].

AST-based technique: Rubicon provides developers a simple application programming interface called AST (Application-Storage-Template) which 3-tiered applications readily map to. In this technique security labels do not have to be specified. End-users express their privacy policies on *data capsules*—a collection of sensitive data that the user groups together.

Specifically, our solution for the program in Figure 4.1 requires the developer to firstly "strip out" the component of the application that sets the policies on data. This part is implemented by Rubicon through the Rubicon trusted interface. Then the developer partitions the program into the following (see Figure 4.2):

1. An *Application* component, containing the executables for setText(), getText(), srchText() and getDocIDs(). Any REST request to this code is now routed via a Rubicon trusted component called *ACL enforcer*. The ACL enforcer ensures that any

[1]Information is said to flow *implicitly* from a predicate used in a conditional statement to all the memory addresses that are affected (written to, or even not written to) based on the conditional decision.

component accessing data capsules are subject to the privacy policies specified on it (Section 4.4.4 expounds on the design and implementation of policy enforcement).

2. A *Storage* component, containing the deduplication executables for dedupPut() and dedupGet(). Any REST request for using deduplicated storage is now routed via the *storage checker* which performs integrity checking on all data capsules that read and written to the storage tier. This allows applications to aggregate data from multiple capsules without increasing the number of labels (Section 4.4.5 explains the mechanism by which robust declassifcation of "mixed" data is in achieved in Rubicon).

3. The *Template* component, which is used for generating code for receiving user input and for securely displaying results across capsules (without mixing content).

It is clear from the AST-modified code in Figure 4.2 that the application in Figure 4.1 is changed very minimally to match the AST design pattern. In practice, because a large number of applications are already modeled according to the 3-tier system design, they will need very limited changes to be adapted to the Rubicon platform. After the developer has specified the above components (note that no security expertise is required), Rubicon executes each of the untrusted components in isolated environments (one per security context) and controls the communication between them through two trusted components, the *ACL enforcer* and the *storage checker*. The creation of data objects with privacy policies attached now has to be policed by the add interface provided by the Rubicon's trusted interface. Also note that when generating the markup for the web application, the HTML generation code will need access to a number of data capsules. The developer can augment the layout generation code with Rubicon's template language (a stripped down version of the Django template language [54]) where HTML code corresponding to each data capsule is generated in isolation from one another.

Eventually the overall execution on Rubicon makes sure that data labels do not get mixed *even if the code accesses data with different security labels* (e.g., deduplication code). We note here that Rubicon's security property *does not depend on the developer correctly conforming to the AST API*. However, following the AST API guarantees most of the functionality for a Rubicon application, e.g., avoiding run-time exceptions when the application accesses data from many users.

4.3 Application design pattern for minimal modifications to existing programs

The Rubicon API is *simple* and does not require that developers consider security policies while programming. What further sets Rubicon and the AST design pattern apart is that partitioned applications can operate on plain-text data from different capsules and yet Rubicon will ensure that data from a capsule can never be leaked from one capsule to another by an application. Specifically, the Storage and Template components compute

Component	API Calls	Rubicon Actions
Application	• POSIX • `put,get_to_storage_chk` • `register_app_interface( wsdl_file )`	Linux syscall API. No compiler/runtime or hardware support required. Rubicon's Storage checker stores a hash of `put` data, and uses the hash to declassify output of `get`. Rubicon uses `wsdl_file` to connect application with presentation layer.
Storage	• `put,get_frm_storage_chk`	Rubicon lets Storage components access *plain text data* from multiple capsules with different ACLs – key to storage optimizations like deduplication. Storage checker uses integrity checking to ensure data isn't leaked across capsules – outputs can be declassified safely.
Template	• Layout Template • `wsdl_function_call( func, data)`	Rubicon uses template to generate HTML views; and ensures that data across capsules are mutually isolated. Rubicon ensures that `data` is sent only to `data`'s capsule-specific Application instance – `data` can thus be declassified safely.

Figure 4.3: The API provided by the trusted components of Rubicon to the AST components.

on cross-capsule data – providing richer optimizations and functionality beyond a simple "application-in-a-box" – while Rubicon implements principled methods to declassify their outputs. Figure 4.3 presents the API provided by Rubicon's trusted components for each of the AST components.

4.3.1 The Application module

The Application component contains all of the functionality that is exposed to users by a traditional application, allowing users to create, edit and compute on data. The Rubicon prototype, discussed in detail in Section 4.4, allows these components to be full Linux applications. Thus any POSIX-compliant application can execute on Rubicon without modifications. For example, applications such as PDF readers and version control systems usually operate on just one document or repository at a time and can be used as-is on Rubicon.

Many cloud-based applications, however, optimize storage across multiple users for caching, deduplication, or replication purposes. Rubicon allows Application components to communicate to such Storage components via HTTP through a simple `put-get` interface.

Further, many applications such as a collaborative text editor (e.g. etherpad) expose their functionality to remote clients over an HTTP-based API. Rubicon allows Application components to do so by registering a Web-Services Design Language (WSDL) [16] interface with Rubicon. Rubicon then uses this WSDL specification to connect the presentation layer of the application with Application instances—thus allowing users to browse and sort *all* of their textpads, or search *all* their textpad for a specific word.

4.3.2 The Storage module

The Storage module comprises of functionality that is not exposed to users, but only to Application components. Storage components requires access to multiple users' data that have different privacy policies, and can implement a diverse set of optimizations: from caching users' data using an in-memory key-value store (e.g., `memcached`) to deduplicating and replicating users' data, for better efficiency and reliability. The Application component communicates with the Storage module through a `put-get` interface.

An AST developer is required to modify the original implementation so that both components communicate over HTTP, allowing Rubicon to monitor all Application-Storage communication.

4.3.3 The Template module

A Template component allows an application to provide functionality that requires access to multiple data capsules, for instance, enabling a user to browse and search through all her textpads for a keyword. Template components are introduced by AST because Rubicon limits each Application component to only execute on one data capsule.

The layout file: Unlike the editor and storage components where developers provide executables, developers are required to provide a *template* file that Rubicon will use to generate the executable of the viewer component. This layout file specifies how users can trigger functionalities like *search* inside individual capsules and how the outputs will be presented to the user (e.g., search results as a list of textpads or a grid of photo thumbnails).

The web page of the display component generated by Rubicon is composed of an outer view and multiple inner views that are inserted into the outer view. The outer view comprises of the application's HTML layout file. Inner views are static HTML content generated by the template processor executing on individual data capsules (as shown in Figure 4.4). Figure 4.2's "Template tier" shows an example of how a developer can insert links to different capsules inside the layout template. URIs generated in the context of a particular capsule will always contain the capsule's identifier in the URI that the ACL enforcer can use to verify permissions.

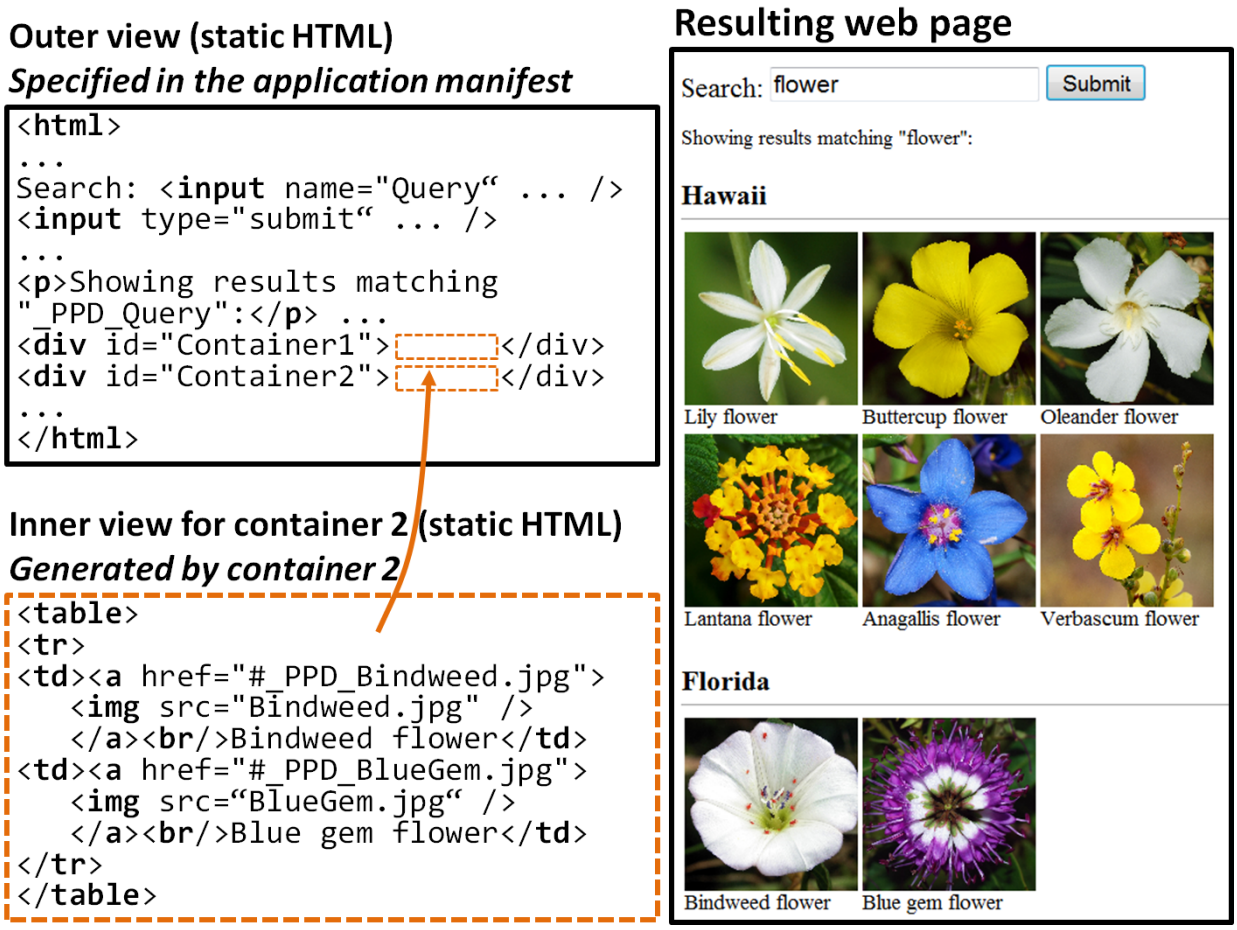


Figure 4.4: An example of the Rubicon Template and how the layout file is used to generate the display code for the user.

4.4 Trusted system components

This section describes the functions and the building blocks of the RUBICON system. The properties of Rubicon are obtained by bootstrapping trust from the trusted code base of the following components. Figure 4.5 shows these trusted components of Rubicon with a shaded background.

4.4.1 ACL editor

Rubicon uses a *trusted interface* through which users can register (authenticate) to the Rubicon system, create and delete capsules and change the capsules' ACLs. This trusted

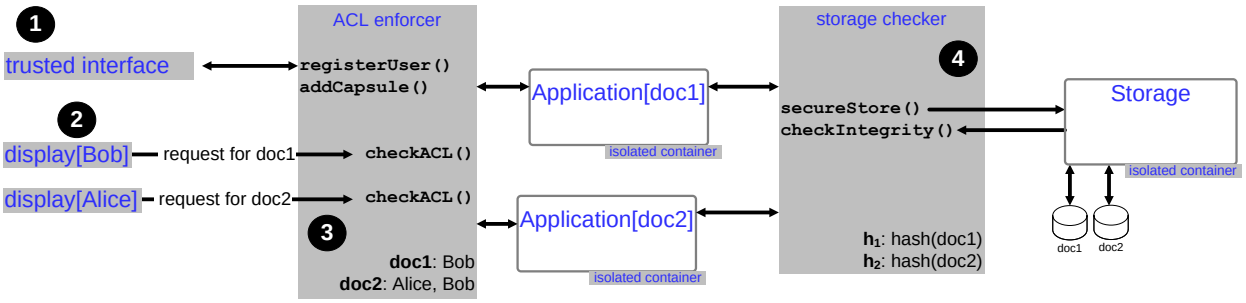


Figure 4.5: The trusted interface is used by the users to create capsules and set ACLs (1). For example doc1 can be accessed by Bob while doc2 can be accessed by Alice. Then every user u is using the display code (generated by Rubicon from the template file) to interact with the application (2). Every HTTP call from the display code $display[u]$ to the an application container is going through the ACL enforcer (3). The ACL enforcer allows the communication (through checkACL()) if and only if $i \in reader[k]$ (or $u \in writer[k]$), depending on whether the HTTP call is reading or writing data. Eventually, every request to the storage container is going through the storage checker, which assures the cryptographic integrity of the answers (4). Specifically data passed from application containers to the storage container goes through secureStore() and data returned from the storage container to the application containers go through checkIntegrity().

interface thus bootstraps the security of the system. An administrator user can interact with Rubicon through its trusted interface, which is also used for user registering/authentication.

4.4.2 Template processor

The end-user accessing the application running on Rubicon primarily interacts with the secure *display code* that is generated by Rubicon using a *template file* provided by a Rubicon application's developer (see previous section). This display code is typically a HTML page and viewed using a browser. The display code $display[u]$ for a user u only offers per-user functionality.

The template processor takes the developer-defined template and replaces the template directives [54] with HTML. For instance, in the example code defined in Figures 4.2, the user wants to see a list of documents she owns before selecting one to work with. The template processor is trusted (has access to all data capsules in order to generate a listing) and creates a layout as specified by the developer's template. Additionally, the Rubicon template processor associates the HTML code for each data capsule with its capsule ID as an attribute. This capsule ID is used to construct the HTTP query string when an Application's API is called. For example, to search for a specific term in an etherpad capsules, the generated HTTP request for a capsule ID `capID` will be `http://www.rubicon.com/capID/searchText?searchStr=+userInput`. Rubicon will launch an Application instance for `capID` to handle this HTTP request.

Finally, the last function that is performed by the template processor is to ensure that the HTML code sent to a client is devoid of dynamic code, i.e., it only contains static content. This is important because a malicious developer could otherwise parse the DOM on the client side using javascript and send it to a third-party server. An alternative option would be use a client side browser extension that sandboxes the web application and restrict communication only to the Rubicon platform. Note that this would break a number of existing web applications as the AJAX calls would be disallowed. In order to support dynamic code execution on the client interface, the user's privacy ACLs should also be applied on the client device. This is dealt with in-depth in Chapter 5.

4.4.3 Containerized execution

Rubicon executes the Application and the Storage components of an AST application provided by a developer inside isolated *containers* that only communicate via Rubicon-mediated components (such as the ACL enforcer and the storage checker described below) as depicted in Figure 4.5. Rubicon uses two types of isolated containers:

- An application container $application[k]$ that can only access *one* capsule k. For example if a user is editing two documents k_1 and k_2 at the same time, Rubicon is using two different containers $application[k_1]$ and $application[k_2]$. The communication of the

application containers to the rest of the system (e.g., to a user u) is restricted through the ACL enforcer;

- A *storage container*, denoted with *storage* that accesses *multiple* capsules. This is where the storage optimization code (such as deduplication, compression) executes. The communication of a storage container to the rest of the system (e.g., to an application container $application[k]$) is restricted through the storage checker.

The ability to spawn containers quickly is key to the efficiency of Rubicon, and a major goal of prototyping was to evaluate the adverse impact of executing a large number of concurrent containers. To minimize container creation overhead, Rubicon maintains container instance that has all popular applications installed, but is not connected to any user's capsule. When the ACL enforcer launches a container for a user, it simply *forks* the entire container and attaches the container to a specific capsule. This minimizes replicated state among containers and also the time Rubicon takes to service a user's request.

Previous projects, such as Linux-CR [73], cryopid and Zap [74] checkpoint the current state of a process tree and revive execution at a later point, either on the same machine or not. While Rubicon could leverage these techniques, we focus on generating immediately running parallel instances of the same container context as quickly as possible. This approach is closer to virtual machine cloning technologies such as SnowFlock [75] which clones instances of the Xen virtual machine.

The state of a container may have hundreds of megabytes of context spread between the persistent file system and the transient physical memory. The memory state contains both kernel and user level data structures. The forking mechanism creates a new container with the same context as the source container. So, we physically replicate as little of the context as possible. This is important for scalability and swiftness in forking. Forking a container involves:

- The file-system context is replicated using a copy-on-write file system (unionfs [143] in our implementation) that quickly creates snapshots of each container. Any subsequent modifications to the file system is written to a container specific scratch space.

- The user-space memory of all processes in the container holds the application specific data structures. Since the page table completely isolates these virtual memory pages between different containers, we employ a method similar to the UNIX `fork`. We introduce a new system call – `container_fork` which creates a new cgroup and recreates the process tree using copy-on-write. The new processes share the same physical memory pages and thus consume no additional memory except for the process specific kernel data structures.

- The file-table data structure in the kernel for each process is replicated for the new process tree and is modified to reference the appropriate file in the replicated file system. A similar approach is used for replicating pipes and sockets as well.

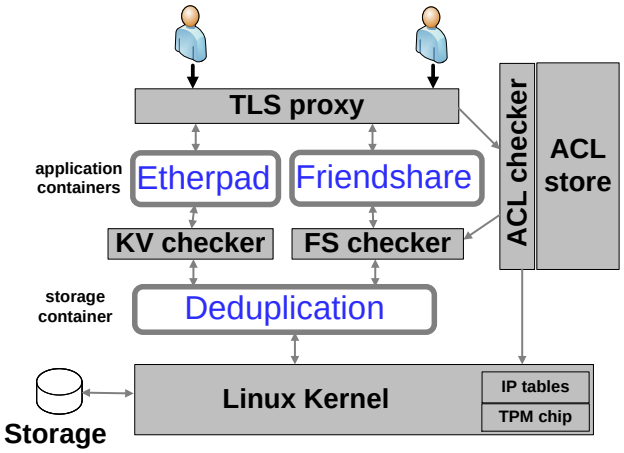


Figure 4.6: Overview of the implementation of Rubicon with two example applications. Application modules execute inside LXC containers, while the ACL enforcer controls network communication using IPTables. The TLS checker routes user traffic to the ACL enforcer if required (to create capsules and update ACLs), while the KV checker and FS checker refer to storage integrity checkers that provide a key-value and file-system interface to application editor instances. The storage integrity checkers prevent the untrusted deduplication component from leaking information among editor instances of different capsules.

- Other process-specific data structures such as the signal handlers, semaphores, etc., are also replicated.

4.4.4 ACL enforcement using capabilities

The set of users that are registered with Rubicon as well as the ACLs $(owner[k], reader[k], writer[k])$ for all data capsules k are stored by the ACL enforcer. The Rubicon ACL enforcer is responsible for dynamically instantiating and terminating containers, based on requests that it receives from a user u.

For example, when a user u wishes to invoke an application function (through the display code $display[u]$) that requires *reading* (or writing) a data capsule k, such as srchText() (or setText()), the ACL enforcer checks if user u belongs to ACL list $reader[k]$ (or $writer[k]$). If this test succeeds, Rubicon checks to see if there is already a container $application[k]$ running for data capsule k. Otherwise it launches a new $application[k]$ container. Overall, the ACL enforcer communicates with the trusted user interface and implements the following set of functions:

1. registerUser(u), for registering new users;

2. authenticateUser(u), for authenticating a user;

3. add(k, v, acl), for storing a new capsule with key k, data v and ACL acl. This function communicates with the storage checker as well, as we will see later;

4. delete(k), for deleting a capsule with key k. This function communicates with the storage checker as well, as we will see later;

5. launchContainer(k), for launching a container for the Rubicon application to execute on capsule k;

6. killContainer(k), for killing an existing container;

7. checkACL(u, k), for deciding whether to allow a request from a user u. This algorithm outputs "accept" if and only if $u \in reader(k)$ (or $u \in writer[k]$ if the request is a write request).

4.4.5 Storage integrity checker

As previously shown, Rubicon's *storage* container hosts untrusted applications' storage components that are allowed to access multiple data capsules with incompatible ACLs. However, all communication between an application container and a storage container passes through Rubicon's storage checker (see Figure 4.5) that prevents information from one capsule being leaked into another capsule.

The storage checker ensures the following invariant: A get(k) request to the storage container should return the *most recent value* assigned to capsule k through put(k, v)—otherwise Rubicon returns an exception. This ensures that even though the storage container could implement caching, deduplication, and other similar functionality on data from different capsules, it cannot mix information among different capsules. Untrusted storage components are thus invisible to application containers.

To ensure the above invariant, Rubicon's storage checker uses integrity checking. The storage checker intercepts a put(k, v) call from an *application*$[k]$ container[2] to a *storage* container and stores a hash $h_k =$ hash(v) of each data capsule's value v (a Merkle tree [93] could also be used, both across capsules and within a capsule if a capsule contains many objects). When some *application*$[k]$ requests get(k) through the storage proxy and *storage* returns a value v', the storage checker returns v' to the editor only if hash(v') $= h_k$. Overall, the storage checker implements the following two simple functions:

1. secureStore(k, v), for computing and storing the hash h_k (if a hash for capsule k already exists, the storage checker stores the new hash by just overwriting the old one). This function is called through a Rubicon REST request from a container *application*$[k]$

[2]Editors can also interact with the file system, in which case each file system call is mapped to put/get calls by the storage checker.

(e.g., when the application wants to edit an existing capsule) or directly by the ACL enforcer (when some new capsule is created);

2. checkIntegrity(k, h_k, v'), for checking the relation hash$(v') = h_k$. If this relation holds, the Rubicon storage checker allows the communication between the untrusted components *application*$[k]$ and *storage*, effectively returning v' as an answer to the respective get(k) request.

Using integrity checking is required because storage applications can access data from all capsules. In information flow terms this implies that storage applications receive the most conservative labels. The storage checker then performs a form of "robust declassification" [100] of data received from untrusted storage. Note that untrusted storage applications cannot affect what data gets declassified since they cannot create a hash collision—robustness of Rubicon's declassification thus stems from its integrity checking mechanism.

Since application containers receive an exception when storage containers misbehave, Rubicon effectively converts information leaks through all explicit and implicit flows in untrusted storage components into termination channels (that are considerably lower in bandwidth [10]).

4.4.6 Extensions

Application containers for clients: The display code only presents static content to a specific user u. The static content is the concatenation of various results that have been produced by executing the application containers over capsules that belong to the user u. For example, when a user wants to do a search for keyword kwd over his documents doc1 and doc2, the display code will just display the output of these searches to the user (e.g., the number of occurrences of kwd in doc1 and doc2). However, web applications have a client component or a browser implementing AJAX requests, that requires processing at the client side. Rubicon's design can support such client functionality by *extending the notion of a per-capsule container to the client side as well.* Rubicon can provide a new container for user u, denoted with *clientapplication*$[u][k]$, which can connect to the application container *application*$[k]$ in the cloud. To create such client isolation environments, we can use browser-based isolation primitives such as Native Client [146] to implement multiple client-application containers within a single browser instance. Alternatively, if the client side is a desktop application, we can use the same isolation mechanisms that we are using in the cloud (e.g., LXC containers).

Aggregating Results from multiple capsules: Rubicon does not support functionality that requires access to the results from different application containers, e.g., sort the outputs of multiple searchText() HTTP requests or output an average. Such a computation by default contains information from all capsules that were input to the computation.

To deal with this problem, Rubicon can be extended to support application containers that read data from many different capsules by using black-box differential privacy (DP)

solutions such as Gupt (Chapter 3 and [98]). We could therefore add a DP (differential privacy) proxy that is analogous to Rubicon's storage checker and ACL enforcer. A DP proxy essentially implements the trusted Gupt system that inputs data from multiple capsules into a container and declassifies the output by perturbing the value (depending upon the privacy budget).

4.5 Security analysis of Rubicon

By default Rubicon sets up the system as if untrusted applications are always limited to one data capsule even in the presence of display and storage components. To achieve this, Rubicon associates an Application component to a specific capsule at launch time and does not change depending on data that it sees during its lifetime. The Storage component is then restricted to communicate only with the Storage Checker that integrity checks and declassifies the Storage component's output on a `get` request. This declassification essentially makes the storage optimizations invisible to the Application components and cannot leak information across capsules. Finally, the Template-based display code is generated to never mix data from different capsules and HTTP requests' data is only sent to an Application component corresponding to the data's capsule.

The Rubicon system thus does not perform dynamic information flow tracking but maps ACL rules into information flow rules on dynamically generated containers. One important implication of Rubicon is that implicit information flows inside an Application component need not be tracked – since Application instances are tied to one capsule, implicit flows within the Application are harmless.

More formally, even with untrusted applications, the overall security property that holds for the Rubicon system can be broken down in the following two ACL-based invariants:

1. (**Secure writing**) During the execution of the system, no user u gets to write to a capsule k that he does not have access to. This is ensured by the ACL enforcer: Every write request originating from $display[u]$ for a specific capsule k is allowed if and only if $u \in writer(k)$;

2. (**Secure reading**) During the execution of the system, no user u gets to read a capsule k that he does not have access to. This is ensured by the ACL enforcer, as above, and by the storage checker: Every read request originating from $display[u]$ for a specific capsule k is guaranteed to return content that belongs only to data capsule k—otherwise the storage container would be able to produce a hash collision.

4.6 Evaluation

This section explores how easily the Application-Storage-Template design pattern can be adapted to existing real world applications. Included in this section is development expe-

riences of porting existing applications to the Rubicon platform as well as the performance impact of the platform.

4.6.1 Applicability of the AST design pattern

Rubicon supports applications where the data owners explicitly decide which other users can access their personally owned data. To determine the popularity of user-initiated sharing, we characterized the top 100 most popular cloud applications [3] based on how information is shared among users. We found that applications with only user-initiated sharing accounted for 52% of the top 100 applications, with examples ranging from storage applications like Dropbox/Google Drive, to online document editors such as Google Docs and Picasa and to real-time conferencing applications like Skype. Even social networking applications like Facebook and Google Plus have several features that can be implemented solely through user-initiated sharing, such as status updates, document sharing, events, and groups.

In contrast to user-initiated sharing, *application-initiated sharing* arises from computing over multiple users' data, and then outputting the results to one or more users. Examples of such applications include recommendation engines in social networks and online retailers, collaborative spam filtering, and data mining tasks where the output necessarily contains information from multiple inputs. In such scenarios, we can incorporate statistical privacy techniques such as differential privacy [29, 98] into Rubicon, as explained in Section 4.4.6.

4.6.2 Development effort in porting applications to the AST design pattern

The flexibility of the AST pattern is discussed using three popular applications that fit the user-initiated sharing model described above, and that together exercise all the components in Rubicon: a) **Git**, a widely-used distributed version control system, b) **Etherpad-lite**,[3] an open-source, real-time document-collaboration software, and c) **Friendshare** [69], a distributed, file-sharing application. These applications are representative of a wider class of applications, as shown in Figure 4.7.

Git can be implemented to be executed solely as an Application instance (with corresponding client-Application instances) to enable cloud-based repository management tools similar to Github (note that only the git application was ported and not the Github application). Both server- and client-side instances of Git are just Linux processes, so porting Git to Rubicon amounted to installing Git inside an LXC container. Unlike Github, however, Rubicon-Git does not implement its own access control mechanism. Any user who has access to a data capsule can launch an editor container and execute Rubicon-Git inside the container to access the capsule's Git repositories.

[3] https://github.com/Pita/etherpad-lite/.

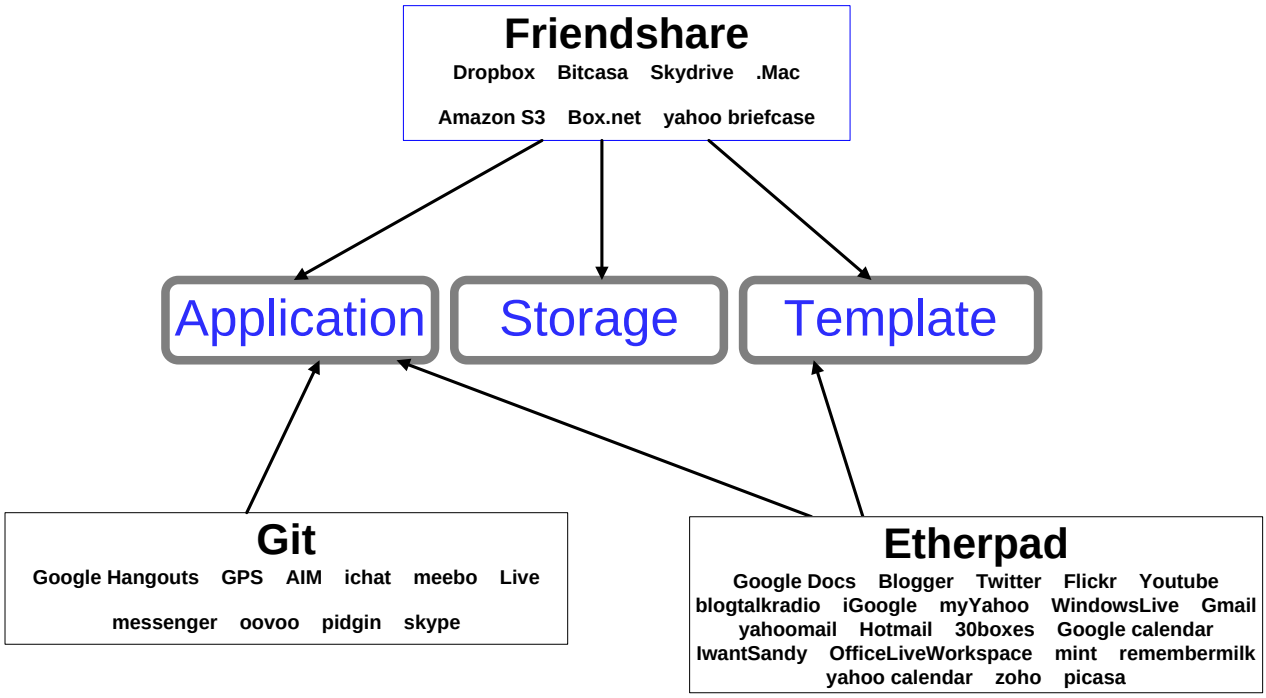


Figure 4.7: Categorization of various applications that use different combinations of the AST components.

Etherpad-lite is a performant *node.js*-based clone of the original etherpad application. This application is more complex to port to Rubicon since it has all three components (Template, Application and Storage) and requires real-time communication among different client-application instances (i.e., the Rubicon ACL enforcer has to create shared application container instances). The code in Etherpad-lite was refactored so that all of its core functionality (creating, reading, and writing to text pads) can execute in it's own address space, while its interface to a database service (its storage functionality) executed in a different address space. A searchText function was added to Etherpad's current HTTP API (about 50 LOC), in order to implement functionality to search in text pads across all the user's capsules through an Etherpad display code. The HTTP responses from each container are sanitized to be free of scripts, all links in the HTML are prefixed with the URI of the container, and inserted into the viewer HTML document by the trusted display code. Starting without any experience with *node.js* applications, it took less than 2 person-days of work to partition etherpad according to the AST pattern and to execute it for the first time on Rubicon.

Friendshare is also a complex application because it uses all three components, but its deduplication component makes its storage more complex than the Etherpad-lite application. We re-factored Friendshare so that its core storage and user interface features form part of

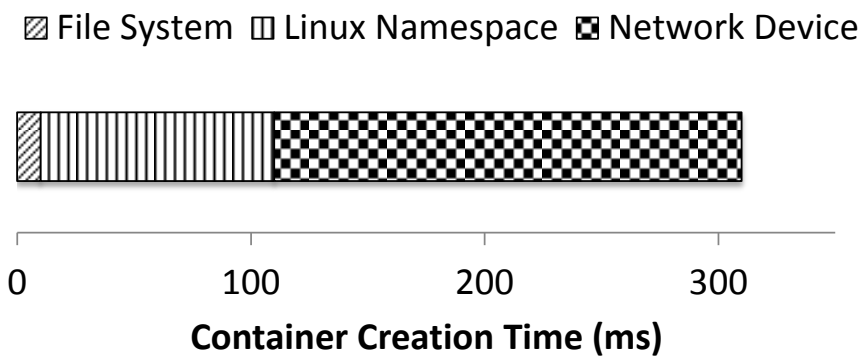


Figure 4.8: Bootstrapping time for initiating a container. Note that the bulk of the container creation time is spent in setting up the Linux namespace and the emulated network device.

its Application component, with shared application instances that allow real-time updates to shared folders, while its deduplication component optimizes storage across capsules. This porting effort took 4 person-days.

In addition to the specific steps outlined above for Git, Etherpad-lite, and Friendshare, there are some general steps that are required to port any new application to Rubicon.

1. Installation: The developer submits a tarball that packages the application container's file system and a configuration file that specifies how Rubicon initializes the application. This container is used as a template for spawning future application instances.

2. Event handling: Rubicon applications must be notified of life-cycle, user management, authentication, and data capsule sharing events. For convenience, the application template specifies how these events will be handled by the application (e.g., via RPC calls or shell scripts). Life cycle events such as Startup and Shutdown are used by Rubicon to notify containers that they have been started or are about to be shut down.

Rubicon provides a user authentication module (ACL enforcer) that is needed to enforce ACLs. As a result, Rubicon applications in Rubicon have their authentication interface and some of their user management features stripped out and replaced with hooks for Rubicon events such as CreateUser, RemoveUser, Login, and Logout.

4.6.3 Performance effects

The overall performance of our prototype greatly depends on the performance of LXC containers. Therefore micro-benchmarks on LXC containers are presented before the performance of Rubicon is evaluated. The benchmarks show that forking performance is fast and LXC containers are a promising method to implement isolation guarantees in practice.

Container initialization overhead: Figure 4.8 shows the delay in creating a new container

Metric	Container Type	LXC Fork	No Fork
Mem (KB)	Small	310.4	52,144
	Medium	309.8	515,992
	Large	315.6	2,060,960
Time (μs)	Small	127.7	169,050
	Medium	125.3	2,000,650
	Large	126.0	6,390,648

Table 4.1: Comparison in container creation overhead with and without Rubicon's container forking behavior. The different container types differ in memory state. Small - 50M, Medium - 500M, Large - 2000M.

from scratch. The initialization latency arises from three major contributors: a) time to create a copy-on-write file system clone, b) time to generate a new Linux namespace (cgroup) and c) time to create a virtualized network device.

The file system clone operation on average consumes about 8ms (on our test system) or 2.5% of the initialization time. The next step is to use Linux's `clone(2)` system call to execute the `init` process of the container that isolates IPC, UTS, file system, network and PID namespaces. This process requires approximately 100ms or 30% of the total boot-strapping time. The rest of the time is consumed by LXC's creation and configuration of the virtual network device that is associated with the container.

We reduce the overall container boot-strapping time by maintaining a set of pristine containers that do not have an application running within them. The application data is mounted and initialized in the container only when required. This will reduce the overall latency of container allocation to $<$ 20ms, i.e., the time for file system cloning and the time to initiate the `init` process in the container.

Container forking overhead: We use the container forking mechanism explained in Section 4.4.3 to reduce the overhead of generating new containers with identical contexts. In Table 4.1, we compare the performance of new container creation with and without Rubicon's forking mechanism. We can see that when we use `container_fork`, the additional memory consumed by the system is irrespective of the memory state of the container. On the other hand if additional containers were started, the memory usage on the system will increase linearly with number of container. Similarly, the time to start a new container is also highly dependent on amount of memory state when `container_fork` is not used. If copy-on-write memory is not used, then the container initialization time increases by 3 orders of magnitude. This is caused by the memory bandwidth needed for allocation. Additionally, when a large number of containers are executing simultaneously, the operating system might start thrashing if copy-on-write is not used. Both forms of container creation assumes that we

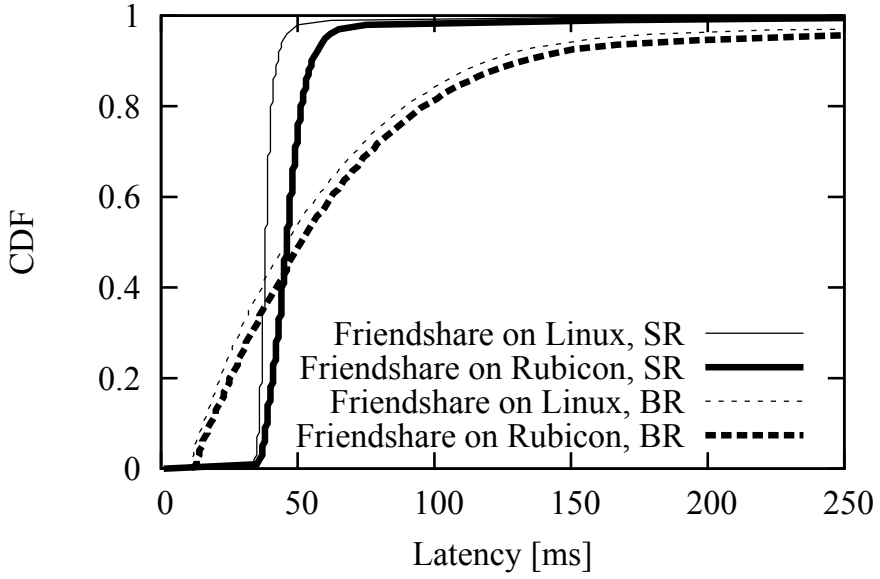


Figure 4.9: Cumulative distribution of request latency for two configurations on small and big requests workloads.

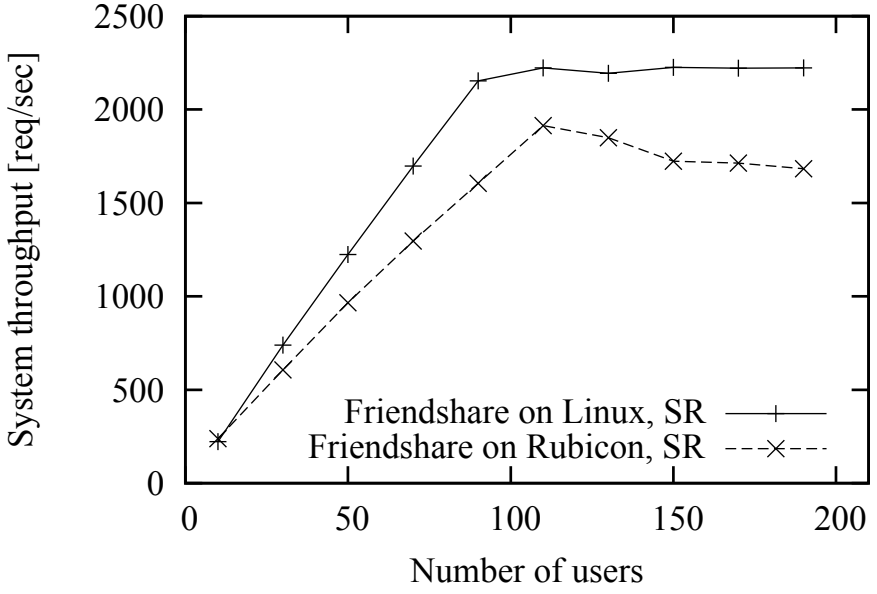


Figure 4.10: System throughput as a function of number of users for small requests workloads.

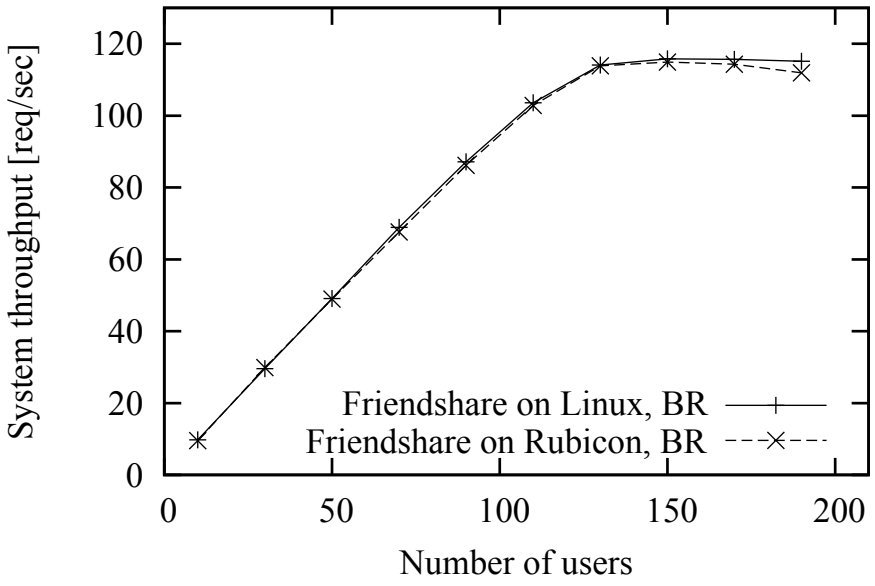


Figure 4.11: System throughput as a function of number of users for big requests workloads.

maintain already initialized cgroups and virtual network devices for new containers.

4.6.4 Effect of Rubicon on applications

Our testbed consists of a Rubicon server machine with 24GB of physical memory and 2 processors with 4 cores each operating at 3.6GHz. Performance metrics are measured on Git, Etherpad-lite and Friendshare. The load generation machines for emulating user workloads are all connected on the same 100Mbps LAN network.

Friendshare: We compare the performance of Friendshare running on an unmodified Linux installation and a Rubicon installation for two types of workloads:

- **Small requests:** Each user repeatedly sends requests to fetch a directory listing. The response for the request is thus cached by the underlying file system at the server. As a result, this workload is context-switch (CPU) intensive.
- **Big requests**: Each user uploads 10KB photos sequentially. After uploading each photo, the user waits for a duration drawn from a Poisson distribution with mean value of 1 second. This workload is bandwidth-intensive.

We evaluate the performance overhead of applications executing on Rubicon using Friendshare as a proxy. We generate traffic from 100 users for 2 minutes and repeat this pattern 10 times. Figure 4.9 shows the cumulative distribution of the latency introduced for

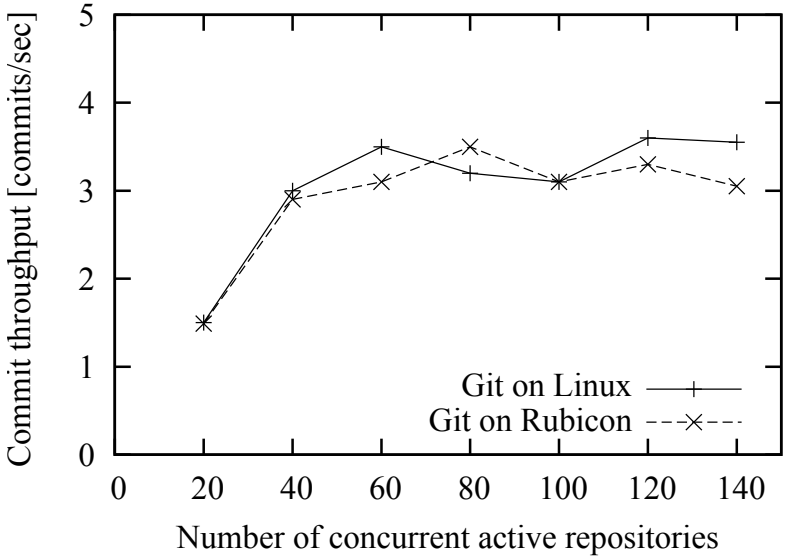


Figure 4.12: Throughput of the Rubicon-modified Git version control server when the Rubicon code repository changes are replayed.

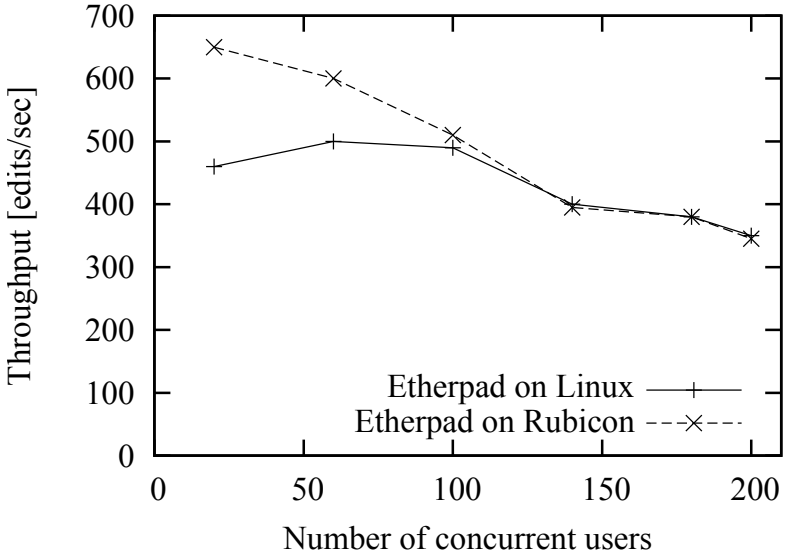


Figure 4.13: Throughput of Rubicon-modified version of Etherpad.

each configuration and each type of workload. The latency for big requests is constrained by the network capacity rather than CPU. We can see that when using Friendshare on Rubicon, the average latency only incurs a smaller overhead for the big requests workload (76ms vs 70ms) compared to the small requests workload (47ms vs 40ms). This could be attributed to the increased CPU contention when we instantiate a larger number of containers (100 in this experiment).

Figure 4.10 shows the change in system throughput (number of completed requests per second), as we vary the number of users in the small requests workload. Irrespective of whether the application is running on Linux or Rubicon, the throughput increases linearly with number of users until the peak throughput is achieved. Figure 4.10 shows that Friendshare running on Rubicon has a 14% reduction in peak throughput (1900 vs 2200 request/sec).

Both configurations reach their peak throughput when there are 110 concurrent users. As more users join the system, the throughput of Friendshare on Linux does not change because the Friendshare server uses a fixed number of threads in a thread pool to serve requests. On the other hand, throughput of Friendshare on Rubicon decreases with increasing number of users since a larger number of isolation containers should be maintained. As a result, there are more context switches which affect the performance of the system. This problem can be partially circumvented by setting the maximum number of running containers to be constant.

Figure 4.11 shows the system throughput when a similar experiment is performed using the big requests workload. The system throughput in both configurations increases linearly as the number of users increase. Once the systems reach their peak throughput, the throughput saturates but holds steady even if the number of users increase. Further, there is almost no difference in system throughput between Friendshare on Rubicon and Friendshare on Linux as the network bandwidth between the client and server machine is the bottleneck.

Git: We run Git $v1.7.4$ on a central server and emulate a centralized version control system with multiple repositories being concurrently accessed by many users. In order to use a realistic workload, we replay changes (sequentially without any wait time) from our own code repository to each of the repositories hosted on the server. In Figure 4.12, we show the overhead of running such a setup of Git on our server, by generating requests from a varying number of committers (each to a different repository).

It is conceivable that Rubicon would have higher overheads with a larger number of concurrently active repositories, as each repository is hosted in an individual container. However, from Figure 4.12, we see that with increasing activity on the server, the increased parallelization increases the throughput (number of commits per second) initially. The throughput quickly reaches a stable throughput rate where we want to operate. We also see that the additional overhead introduced by Rubicon over and above a canonical installation of Git on a Linux server is a maximum of 15% since the workload is network I/O bound (the overhead is introduced by network virtualization). Thus, even if a large number of repositories is concurrently active, Rubicon would only add small overhead to the overall functioning of the

system.

Etherpad-lite: We evaluate Etherpad-lite by measuring the maximum sustainable throughput of document edits. Each edit consisted of a getText() followed by a setText(*text*) operation. The size of the text *text* was drawn from a Gaussian distribution with mean 5KB and standard deviation 1KB. The server-side code of Etherpad-lite is written for the node.js framework which is event driven and single threaded.

We see in Figure 4.13 that because Rubicon runs separate instances of Etherpad-lite, the platform introduces parallelism in the Etherpad workload (which is inherently a single threaded application) and sometimes achieves higher throughput than running a single instance of Etherpad outside of a container.

4.7 Related work in providing privacy guarantees

Systems such as BStore [20] and CryptoJails [126] argue for the decoupling of the storage component from the rest of an application: users entrust their files to trusted storage, which enforces policies on their behalf. Rubicon cares about not only data at rest, but also arbitrary application modifications and sharing, which goes beyond the goals of these systems.

Storage capsules [17] share Rubicon's goals: contained execution of untrusted code while sensitive capsules are manipulated in the clear. However, storage capsules take over a machine while execution occurs within a container, suspending network output, and turning all non-capsule related computation discardable. Although defensible in a client, this approach would counteract the resource sharing necessary in a cloud infrastructure. Similar to storage capsules, policy-sealed data [122] use attribute-based encryption, allowing decryption only by nodes whose configuration matches a specific policy.

Rubicon's application partitioning approach is reminiscent of CLAMP [109], which also rearchitects a web service to isolate various clients by refactoring two security-critical pieces into stand-alone modules: a query restrictor (which guards a database) and a dispatcher (which authenticates the user). Although along the same direction as Rubicon, CLAMP stops short of defining how to enable controlled sharing of user data through untrusted code. Similarly, Secure Data Preservers [60] isolate the web-application logic that operates on sensitive data behind an agreed interface, and enforces isolation between the application and that logic. Unlike data preservers, Rubicon does not require the definition of per-data-type interfaces, but requires the execution of an entire application within containers, which simplifies application porting and encourages adoption.

Finally, *Hails* [45] and *self-protecting data* [21] are two recent projects related to Rubicon. The Hails system provides a programming framework that augments the Model-View-Controller application design pattern to perform information flow tracking. It offers an API to the developer for constructing applications that cannot violate user policies when executed on a trusted server. This constrains the developers to use a specific programming language as well as laying the arduous task of labeling every member variable in the data structures either on the user or the developer. These applications also have to give up communication

privileges in exchange for access to user data, causing unexpected runtime exceptions for the user. Self-protecting data achieves isolation through hardware-assisted information flow tracking and a security policy component that is separate from the rest of the operating system. Being based on dynamic information flow tracking, neither Hails nor self-protecting data protect against implicit information flows.

Most importantly, both Hails and self-protecting data achieve privacy control through an "app-in-a-box" model, where an application is confined to one security label. Rubicon, on the other hand, allows application components to access plain-text data belonging to different labels and yet achieves information flow control through robust declassification of display and storage outputs.

Computing on encrypted data: Conceptually, Rubicon is attempting to approximate computation on encrypted data. Cryptographic techniques have provided alternatives [43, 44, 112]. However, much research is still needed to make such techniques practical [76].

Code attestation: Trusted Computing and code attestation [88, 89, 121, 127] technologies are an important building block allowing Rubicon to offer more transparency to the general public. BIND [127], Flicker [89], and Trustvisor [88] propose to isolate the execution of a small Piece of Application Logic (PAL), to ensure either secrecy or integrity. However, the PAL must contain only CPU instructions and cannot make system calls or access system resources. Rubicon builds on top of such mechanisms light-weight virtualization to more easily support legacy applications with system calls.

4.8 Summary

Rubicon is a platform that allows existing multi-tiered services provide users with a privacy guarantee regarding the entities with which the user's data is shared. Rubicon makes the following contributions:

1. **Design framework:** It defines a simple privacy-preserving application design pattern (AST) that introduces minimal changes to a 3-tier system. Applications architected with this design pattern are privacy-preserving by construction without requiring significant security expertise from developers.

2. **Ease of use:** The Rubicon platform infers information flow rules based on user-specified ACLs and enforce these rules on all inter-component communication in a partitioned AST application.

3. **Practical prototype:** We extend existing container based isolation mechanisms in Linux to rapidly create copy-on-write clones. Using a prototype implementation, we show that Rubicon applications exhibit performance overheads of only 2.5–15%. This prototype requires no hardware (as opposed to [21]) or compiler/run-time support.

4. **Low developer effort:** Finally, the effort required to port existing applications to the Rubicon platform as well as in writing new applications.

Chapter 5

Privacy with Internet (dis)connected applications

Mobile application distribution mediums, such as Apple's "App Store" and Google "Play", have been instrumental in the adoption of mobile computing devices, allowing otherwise unknown publishers to sell apps to millions of smartphone and tablet users around the world. Today, these extremely personal devices are used to perform telephony, messaging, financial transactions, gaming, and many other functions. While the popularity of the app store model attests to the benefit and utility of allowing third-party apps, these applications introduce a host of privacy and security risks [38].

A number of different security mechanisms have been used to provide data privacy. For instance, Android's permissions model [1] is an example of *application-centric* security. Android has a static capability-based system where users must decide at installation time whether to grant permissions (including network and device access) or not (and forgo use of the application).

As was discussed in Chapter 4, information-flow tracking systems [27, 152] are more expressive than static capabilities and provide *data-centric* security, but require considerable sophistication on the user's part to translate security policies into a lattice of labels. Moreover, they often require applications to be modified with security label assignments so that they do not crash with security exceptions. TaintDroid [35] is an example of an information-flow tracking system for Android, which modifies the Dalvik virtual machine to propagate taint through program variables, files, and IPC messages. While it can track some unmodified Android apps, it cannot track applications which use their own native libraries. Further, information leakage through implicit flows [66] cannot be restricted.

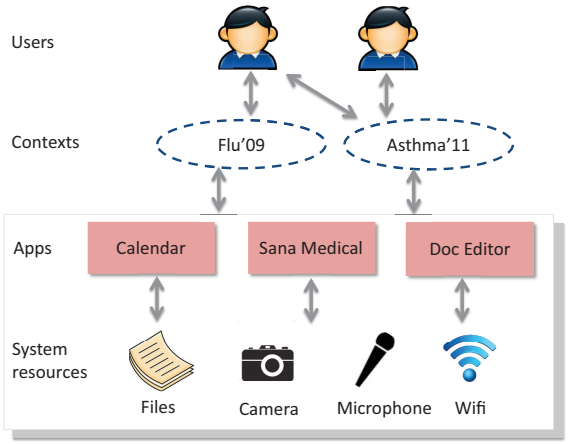


Figure 5.1: Traditionally, security policies are expressed in terms of permissions on applications or security labels on system-level features. This makes it hard to capture users' intentions that stem from high-level, real-world contexts, and lead to either static, inflexible permissions as in Android or sophisticated policies and implicit information leaks as with TaintDroid.

5.1 Data isolation in client devices

Security systems can only be effective if they present a security model that matches the way users reason about privacy in their real-world interactions. This chapter introduces BUBBLES, which allows a user to define a digital boundary (called a *bubble*) around the data associated with a real-world event. For example, the event might be a meeting for a work project or a birthday party. Goffman [46] proposed that people present different *faces* of themselves depending upon the social context. Bubbles implements this sociological aspect of privacy in a digital setting. Similar to attaching access control (ACLs) permissions to files in a file system, the user specifies her privacy requirements for the bubble as a list of people, taken from her contacts list, who have access to the bubble. Different sets of people might have access to different work projects, or to different birthday parties. Bubbles preserves this privacy model across all the user's multiple devices and the cloud thus providing *context-centric* security which isolates all data between bubbles. Bubbles translates these simple user-centric access control rules into the more complex information-flow rules in the underlying system. We have also developed a simple application design model to enable app developers to provide extensive functionality for the user without requiring either the user or the app developer to worry about privacy enforcement. Bubbles effectively factors out privacy features from applications, and puts privacy control into the hands of the user in a natural way.

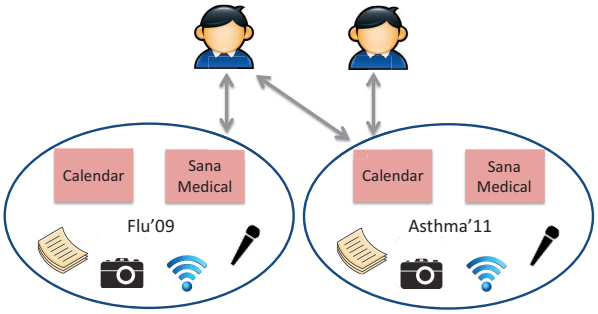


Figure 5.2: Bubbles represent real-world contexts that are potentially shared among multiple users, and around which users' data automatically clusters. Users' privacy policies can then be directly represented as an access-control list on bubbles. Applications and low-level peripherals then exist solely to provide functionality and cannot affect who sees the contents of a bubble. Bubbles are thus implemented as tightly constrained execution environments (like a virtual machine, but much lighter weight), and require applications to be partitioned to provide the functionality associated with legacy applications.

5.1.1 Android permissions considered insufficient

Android is an operating system that includes a modified Linux kernel together with standard user space sub-systems, and is targeted toward mobile devices such as smartphones and tablets. Android provides more than 100 permissions that an application can request at installation time, which a user must explicitly approve or deny. Although, Android warns users by marking some permissions as dangerous, "93% of free and 82% of paid applications request at least one dangerous permission" [39]. The sheer number of permissions being requested causes users to be indifferent about security. Moreover, we find that Android exposes permissions that are foreign to even sophisticated computer users, such as `MOUNT_FORMAT_FILESYSTEMS` (which allows applications to format file systems on removable storage). By asking users questions at the wrong level of abstraction, Android leaves users' privacy in the hands of untrusted applications. Finally, since permissions in Android are statically enforced during installation, applications have these permissions forever. Static permissions allow any application with microphone access, for example, to record a user's voice without her explicit confirmation.

5.1.2 Flexibility of the Bubbles security paradigm

We analyzed 750 of the top free and the top paid (375 of each) Android applications from the Google Play store to determine how their functionality relates to users' privacy. On one end, from a privacy point of view, are applications that provide functionality that is not tied to a user's real-world identity. Examples of such applications include flashlights, games, wallpapers, dictionaries, news sites, and browsing for reviews and recipes among others. Such applications can run inside an "anonymous" bubble where the users are expected to

not enter sensitive information, and can move data to and from arbitrary locations into the anonymous bubble. We find that 45.6% of the free and 45.3% of the paid applications fit this model.

The second category of applications are where users actively create data (we assume this data is sensitive) and then *explicitly* share this data with other users. Applications that allow storing, editing, and sharing of documents (in formats that range from simple text and images to even audio and video), and real-time communication applications that use SMS, MMS, voice, or video fit this category and account for 47.4% of the free and 52.3% of the paid applications. The common feature of these applications is that users can specify for a given blob of data who they want to share it with – in essence an Access Control List (ACL) for each data blob – and the key insight behind Bubbles is to tie these arbitrary blobs of data to a real-world context.

The remaining 7% free and 2.4% paid applications perform what we term *application-initiated sharing*, where functionality, such as a recommendation service, requires that users give up their personal data to the application and the application mixes information from multiple users to generate new suggestions or insights. ACLs do not capture the privacy requirement here, because a user has to give up her data to the application, and alternate definitions of anonymity, such as differential privacy, are required to guarantee that a user cannot be singled out from a dataset by an untrusted application. Most social networking applications include features that implement explicit communication of data, which can be integrated into Bubbles, while features that initiate sharing through aggregate analytics require an anonymizing proxy to enforce privacy through differential privacy. Note that in this chapter, we only discuss the client-side implementation of the applications; the server side of Bubbles can be provided by the Rubicon system described in Chapter 4.

5.2 User Abstraction

Bubble: The core hypothesis of this chapter is that users want to work in *contexts*, where a context encapsulates information of arbitary types— be it audio, video, text, or application-specific data— and is often tied to a real-world event involving other people. We call such light-weight contexts that have data and people associated with them Bubbles. Applications just exist in each bubble to provide functionality, and any data that a user accesses will trigger its corresponding application.

Users' privacy policies are inherently tied to such contexts and thus are best stated in terms of Access Control Lists (ACLs) of contacts for each bubble. If the users are effectively broadcasting information (as when they browse websites or public forum) they want to be aware of this and act accordingly.

On creating new bubbles: A bubble is effectively the minimum unit of sharing, because when all apps that act on data inside a bubble are untrusted, they can mix data arbitrarily among files that exist within a bubble. The implication of this is that sharing even a part of the data in a bubble is equivalent to sharing any data from the bubble.

As a result, we recommend that bubbles be tied to very light-weight contexts in order to facilitate flexibility in future re-sharing decisions. A coarse classification of all personal data into, say, a "Home " bubble and a "Work" bubble will lead to violation of privacy guarantees when the user moves even a single file across this Home-Work boundary. On the other hand, a light-weight event could be a single meeting, or even only a part of a meeting (e.g. the technical discussion as opposed to financial discussions), and putting these light-weight contexts into separate bubbles allows a user to share these smaller units of data independently. In the example above, all developers may be included in the technical discussion, but only the program managers may have access to the financial details of the project.

Navigating the foam: We call the collection of bubbles visible to a user, their *foam*, which replaces the conventional user-visible file system. The Bubbles system only supports a flat *foam* of bubbles without hierarchical order. The system executes bubbles inside independent, mutually isolated containers, and does not support nested bubbles, in order to prevent complications that arise in constraining untrusted applications. Since untrusted applications can operate on the data in an arbitrary manner inside a bubble, they can mix information among all data items in sub-bubbles. Sharing any data item to a new person will thus leak information from potentially all sub-bubbles to the person. Traditional systems propose fine-grained information flow analysis to control how applications mix information, but tracking implicit flows at run-time leads to considerable performance penalties. As a result, we eschew fine-grained information flow tracking in Bubbles, and build Bubbles around the basic primitive of an isolated container.

While bubbles cannot be nested, users can assign an arbitrary *tag* to a collection of bubbles, e.g., to group bubbles as belonging to some longer-term project, and can even overlay a hierarchy on the underlying foam of bubbles. Further, the system tags bubbles with time, location, nearby contacts, and other contextual information that may help the user identify or index the bubble for future reference. For instance, a user can view her bubbles as a time-line to give a calendar view, or by geographic coordinates overlaid on a map view.

Staging Area: Users often create data that is not immediately associated with any existing bubble or tag, for example, a phone-camera photo or a web-page downloaded for future reading. In such cases, instead of forcing the user to assign this data to a bubble, the system automatically assigns the data item to a new bubble and assign location and time-based tags to the bubble. This ensures that the user has flexibility of copying such data items into any (even multiple) bubbles later on. This can be used to implement a Photo Gallery application for example, allowing browsing of images without mixing information from one image to another. One fallout of the staging area is that the system will have a *lot* of bubbles, motivating the need for a very light-weight implementation of containers in Bubbles.

Usage Flow: We now use the example shown in Figure 5.3 to illustrate how a user works in a context-driven rather than application-driven manner. We argue this adds little cognitive

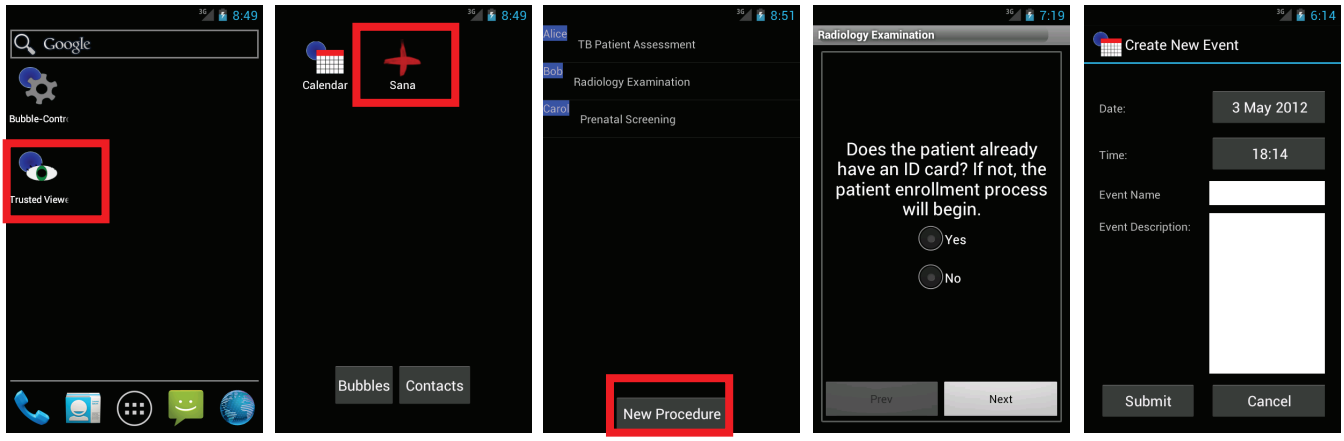


Figure 5.3: Usage Flow in the Bubbles system: User's Home screen shows trusted system applications to manage Bubbles and to launch the Viewer. The Viewer allows a user to see all installed applications, such as a Calendar or the Sana medical application. Clicking an application in this mode takes the user to browse cross-bubble data, i.e. all data attached to Sana or the Calendar. Within the View mode of an application, the user can initiate new data creation; either in a Staging area (i.e. for which the system assigns a unique bubble), or by first using the Bubble service to transition into a bubble and then going to the edit screens for Sana or the Calendar (the last two screens on the right).

overhead to regular operation and that it is worth the price of privacy.

One usage flow could be to start with the trusted *Viewer* application that allows a user to browse data from her entire foam within a single bubble, classified either by application, such as Sana or Calendar in Figure 5.3, or by data type, such as images. Clicking on Sana takes the user to a listing of all medical records classified by bubble name (each patient is stored in a separate bubble). The Viewer app is discussed in more detail in Section 5.4.

Clicking on the "New Procedure" button takes the user to the staging area, where the user can enter data related to a patient's visit. At the end of the procedure, and before the user moves back to the viewer mode, Bubbles prompts the user to enter a new name and tag for the bubble and other contacts who this bubble is shared with (e.g. a remote doctor). In case of an existing patient, the record can be assigned to an existing bubble. The last image on the right shows that a user, while she is in a bubble, can switch among applications (from Sana to the Calendar here) while staying within the same bubble.

Apart from the Viewer, the two trusted applications that a user interacts with in Bubbles are the Bubble Manager and Contacts Manager. Both the Bubbles and Contacts applications are available on the Home screen (which can also include some recent and favorite bubbles for easy navigation) to navigate to an existing bubble or to create a new bubble and invite contacts into it.

5.3 Bubbles system design

In this section, we describe the design of our Bubbles prototype implemented on top of Android. All applications are available to each bubble, with all but a few permissions (explained in Section 5.3.3). Applications effectively execute as though each bubble was an entire OS installation.

5.3.1 Isolation between Bubbles

In order to provide file-system state isolation between each bubble, we choose an approach similar to Cells [7]. We mount a unioning file system that uses the base operating system files as a read-only copy, and a read-write scratch directory to hold any file system modification created by applications in a bubble. Note that OS files cannot be modified by any application running inside a bubble, and that scratch directories are maintained on a per-bubble basis to prevent sharing of file-system state between bubbles.

Control groups [92], supported on both Linux and its smartphone fork Android, are used to provide UTS, PID, IPC and mount namespace isolation, to further ensure that applications cannot communicate with each other apart from via the Binder IPC mechanism. We also make use of the same techniques used in the Cells system [7] to provide device isolation, i.e., because we could potentially have multiple applications accessing the same device resources, it is important that devices understand the namespace abstractions.

We also enable all Android middleware services that allow persistent state to separate data between different bubbles. For instance, `SQLiteOpenHelpers` respond to `getReadableDatabase()` or `getWritableDataBase()` calls from applications with a bubble-specific instance of a database. The SD card and preferences are also virtualized in a similar manner.

5.3.2 Copying data between Bubbles

A user may copy data from one bubble to another, e.g., when an image from the staging area is copied into an existing bubble. Such copying requires applications in the receiving bubble to be able to assimilate the incoming data structure instances with existing ones. Because the Bubbles system is agnostic of application data structures, the applications register call-back functions for handling inter-bubble data transfer.

The Android IPC mechanism is modeled upon the OpenBinder functionality in BeOS [18] wherein a `Binder` device mediates communication between different processes. In Android, this communication can be effected by using the `Context.registerReceiver()` routine to register for incoming messages (called `Intent`s in Android). Bubbles's modified `Binder` driver implementation automatically modifies the `IntentFilter` to restrict messages sent to the application.

The source bubble initiates data transfer by sending a message to the `Sharing Service` implemented as part of Bubbles. In our prototype, this generates a trusted prompt asking for

user confirmation about the application-initiated data share. (This prompt can potentially be removed by replacing the `Sharing Service` with trusted and isolated widgets similar to the access control gadgets [115] architecture). Finally, the applications can transfer data in any serialized format it prefers such as Google Protocol Buffers.

5.3.3 Intuitive permissions model

In Bubbles, instead of the applications statically requesting resource permissions at install time, permissions are automatically inferred from the access control rules placed on the bubble within which the application is executing. To achieve this, Bubbles relies on explicit user input and on virtualizing the Android resources among different bubbles. Most permissions fall into *three* broad categories.

Explicit user decision: Users are provided with a trusted UI to explicitly enable 7 permissions, in order to input audio, video, location, and contacts into a bubble. These resources have the common feature that users are familiar with these concepts outside of a computing device context and can thus make intelligent decisions about when to share them within a bubble. Writing to the contact list is however limited to the trusted address book application.

Per-Bubble resources: Internal and external storage, logs, calendar, application caches, history, various settings like animation scale and process limit are all low-level resources that need not be exposed to users. Bubbles allows all applications to have access to the 27 permissions that control these resources, while maintaining a unique, per-bubble copy of each resource (e.g. isolated folders in storage).

Concurrently shared resources. Communication resources like telephony, wifi, and internet access are all shared among various bubbles. Hence Bubbles enables all 17 communication-related permissions to all applications in a bubble, but with firewall rules to ensure that all communication follows ACL rules specified by a user. Hence applications in a private bubble can only communicate with Bubbles servers (or only send SMS messages to a contact who can access the bubble), while an application in an anonymous bubble can talk to arbitrary servers. While not relevant for privacy, additional rules can be imposed to prohibit communications that cost money.

5.4 Developing applications with Bubbles

Simple programming model for developers: As shown in Figure 5.4, legacy application functionality can be partitioned into two components: *Editor* and *Viewer* modes. The editor mode includes most user-facing functionality that one typically associates with a legacy application, e.g., adding and editing patient records in Sana or new events in the Calendar. Bubbles then ensures that there is a unique editor instance for each bubble so that untrusted editor code will only ever see data from one bubble.

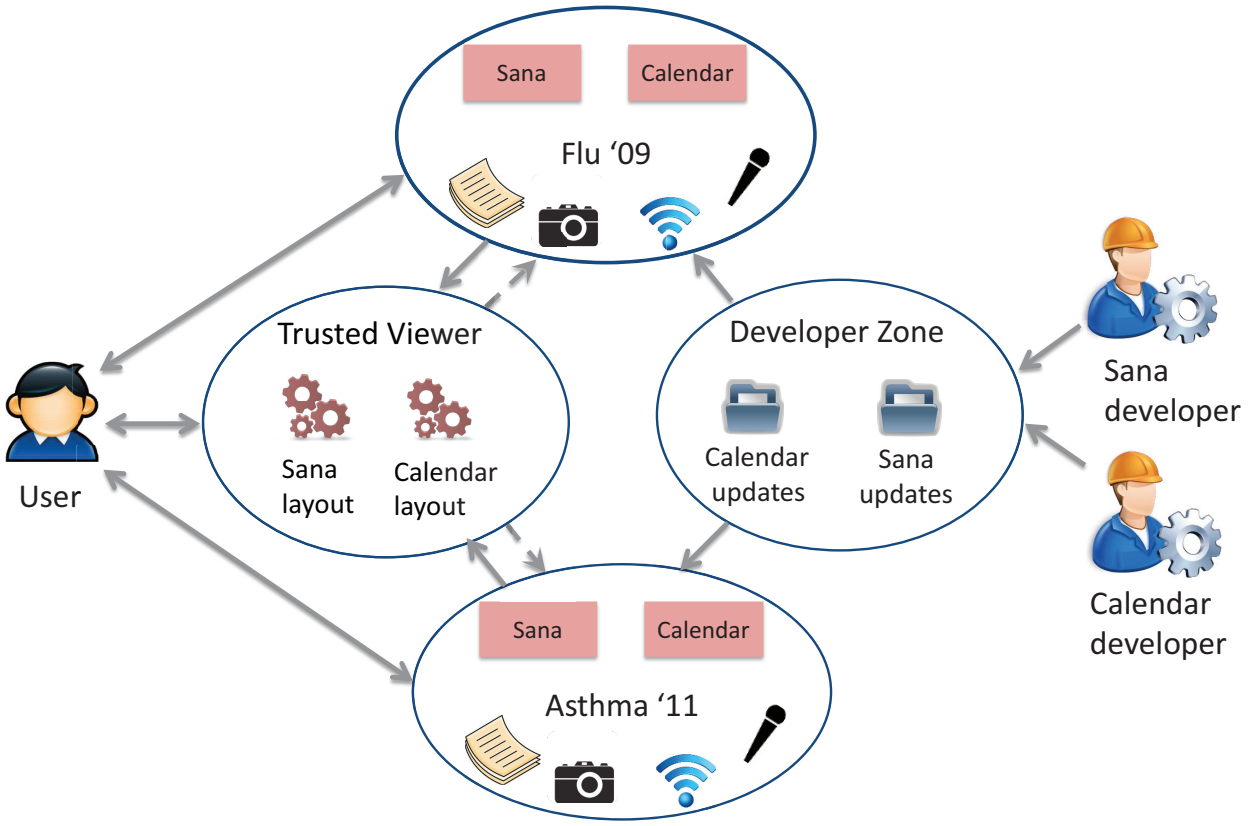


Figure 5.4: Applications in BUBBLES: Most application functionality is included in its *editor* component with one editor instance inside each bubble (e.g., inside Flu'09 and Asthma'11 bubbles). A *Viewer* bubble provides cross-bubble functionality by combining trusted code that receives and processes data from multiple bubbles and a layout file specified by the application statically that determines how such data is laid out. Finally, Bubbles provides developers with their own bubble to send in application updates that a user's personal bubble can only read from and never write to.

The viewer mode of an application is required for the user to be able to browse data from multiple bubbles, e.g. medical records for all patients in Sana. There is thus only one viewer instance on a client device (or one per user). The viewer is responsible for capturing the user's intentions when she uses the application's interface (e.g. clicking a patient record or searching for a particular illness), and forwarding the resulting query to individual bubbles. This requires a message to be passed from the viewer to one or more editors to convey, for example, a specific data item to be opened or a specific term to be searched for. Since the viewer accesses cross-bubble data and then generates this message, an untrusted viewer can encode information in the message. Thus, we only allow fixed builtin *trusted viewers* to prevent information leaks among different bubbles.

Cross-bubble functionality: To enable application-specific functionality in the viewer

mode, an application specifies a layout file and a call-back function at install time. Applications register themselves with the trusted Viewer Service by sending an intent containing 2 XML files: `ui.xml`, which informs the Viewer Service how to display the layout of the application's viewer mode (our current prototype supports the LinearLayout ViewGroup and so far the only supported Views are `ImageView`, `TextView`, `GridView`, `ListView` and `EditText`.); and `app_data.xml`, which contains basic application information (name, icon, etc) together with a callback to be called whenever any entry in the viewer mode is acted on by the user. Sana enables a user search (for any text string) by registering a "search" button as an action item in `app_data.xml` and using the call-back function to receive the search string as a message. The user can control which bubbles the message is sent to.

Preliminary evaluation: We have so far ported two applications— Calendar and Sana —without requiring substantial code changes. The editor mode had to be changed to call Bubbles wrappers around Android services (e.g. SQLiteOpenHelper replaced by DBSwapper), a call-back function implemented to parse messages from the viewer component, and XML files that indicates how the viewer should display calendar or medical records from multiple bubbles. We needed 500 lines of code to port Sana, including all wrapper classes, and implemented the Calendar app in about 800 lines of code.

Developer zone for ads and updates: We introduce Developer Zone as a storage area to support advertisements and software updates (Figure 5.4). Its key feature is that it can be written to by developers, but only read by application editor instances in each bubble.

Bubbles is complementary to several prior works on privacy-preserving advertising [41, 48,137] – basically, Bubbles can be readily used to provide a secure client-side implementation for these privacy-preserving advertising systems. A dedicated ad retrieval bubble retrieves a set of ads from the ad network, based on information (e.g., a broad interest category) a user explicitly shares with the ad retrieval bubble. An ad selection bubble (i.e., a viewer service) can potentially perform user profiling on sensitive, fine-grained personal information or behavioral traces of the user; it reads the ads retrieved by the ad retrieval bubble, and selects and displays the most relevant ad.

Software updates, as shown in Figure 5.4, requires new information from the developer to be made available to the application. The server conducting the updates (e.g., Google Play) will be able to write and read from the Developer Zone, but for the applications' editor instances, this will be a read-only area. An application can provide a button for the user to indicate when she wants to check for updates and then, the application just uses the Bubbles API to check, read and apply the updates from the DevZone.

5.5 Related work in user context-based privacy policies

The idea of isolating applications and data on a mobile based on the user context has been widely explored in literature [58, 123]. Bubbles extends existing literature by providing developers with an easy to use design pattern for developing fully functional applications that abide by the users' privacy policies at the same time.

Helen Nissenbaum argues that privacy is closely linked to prevailing social, economic, and judicial norms, i.e. the prevailing social context [104]. We agree with this premise; in fact, Bubbles system allows a user to be clear about what she has shared with whom, and thus will form a basic primitive for the system Nissenbaum envisions. We use the term context to mean any small event (a meeting, a browsing session) that generates some arbitrary data.

MobiCon is a system that provides context detection services to context-aware applications and implements the detection algorithms in energy-efficient ways [77]. However, Bubbles uses the term contexts to indicate a real-world event that creates data that has some access control attached to it. Bubbles could, in addition to user input, use MobiCon as an underlying service to better infer automatic tags, or even the start/end of a bubble's lifetime.

The Cells [7] system maintains parallel Android execution environments running unmodified Android applications, where it creates "virtual phones" that are completely isolated from each other. It makes devices namespace-aware by introducing wrappers around drivers, as well as modifying the device subsystem and the device driver itself. Bubbles uses the concept of 'virtual phones' proposed in the paper to generate bubbles since they serve the same purpose. In addition to the Cells infrastructure, Bubbles also performs permission inference using the peripheral device virtualization and provides an API for untrusted applications. In contrast to Cells, where a few virtual phones are envisioned to execute all the time, Bubbles will create a large number of bubbles that need to be managed using trusted indexing services.

There has also been work to reduce access permission prompts by providing trusted widgets called 'Access Control Gadgets' that an untrusted application can insert to get access to privileged resources [115]. These gadgets are isolated from the application and their integrity ensured. ACGs can thus be used in Bubbles for the user to convey a carefully chosen set of permissions. Unlike Bubbles, once an application has access to the data, ACGs do not allow a user to enforce context specific privacy policies on the data.

TaintDroid [35] is closest in spirit to Bubbles. As explained earlier, TaintDroid ensures that private sensitive data sources are not exposed by propagating taint through files, variables and IPC messages. However, information leakage is possible in TaintDroid through implicit flows. Further, because information flow control requires source code to be annotated with security labels, TaintDroid disallows application-specific native libraries. Bubbles guards against both of this by isolating all state between bubbles.

5.6 Online advertising as a case for disconnected operation

The rest of this chapter analyzes the design of a prototype system – MobAds. This system is used to explore to what extent today's advertising systems can afford to offer ad prefetching to mobile clients. We start by considering the problem of predicting for how long a certain user is likely to use phone applications in a given time interval. This prediction

gives an indication of how many ads the client will be able to consume in the future. The prefetching of ads is considered to be a given for most privacy-preserving ad system today. Further we find that the client device will have significant energy benefits by prefetching advertisement ahead of time.

The consumer draw of free mobile apps over paid apps is prevalent in app stores such as the Apple AppStore, the Google Play market and the Windows Phone Marketplace. The Google Play market, for instance, reports roughly 2/3 of their apps as being free [8] of which the large majority (80% according to a recent study [78]) rely on targeted advertising as their main business model. With a total of 29 billion mobile phone applications downloaded in 2011 alone [5], the mobile advertising market was valued at 5.3 billion US dollars in 2011 [55].

While the use of in-app advertising has enabled the proliferation of free apps, there are 2 major issues: a) the advertising network captures and utilizes (and often resells) a sizable amount of personal user information in order to tailor the ad for the user and b) the fetching and display of the advertising contributes significantly to the application's energy consumption.

For popular Windows Phone apps we find that on average ads consume 65% of an app's total communication energy, or 23% of an app's total energy. Previous studies on Android apps [63, 110] have shown an overhead of even 45% for a game such as Angry Birds. This excessive energy cost for serving ads is partly due to user tracking (e.g., GPS monitoring for location-based ads), but more significantly to network utilization for downloading ads. Mobile apps usually refresh their ads every 12–120 seconds [47]. Just downloading an ad takes no more than a few seconds. However, when an ad's download has completed, the 3G connection stays open for an extra time, called 'tail time', which depending on the network operator, may be, for instance, 10 (e.g., Sprint 3G) or even 17 (e.g., AT&T 3G) seconds [96]. The tail time alleviates the delay incurred when moving from the idle to the high-power transmission state, but at the cost of energy, so-called 'tail energy'. This is in fact what causes the ads' large energy overhead.

While battery lifetime represents a major limitation for smartphones, memory capacity is still experiencing reasonable improvements [56], and can be used to lower the communication costs. Extra memory space can be used to prefetch ads in bulk, periodically or perhaps when network conditions are favorable. The mobile platform can then serve ads locally, until the budget is exhausted or the cache is invalidated (e.g., ads have expired).

The other major issue in mobile advertising is the privacy risks involved in the process. Recent work on privacy-preserving advertising [41, 49, 50, 59, 138] also implicitly assumes prefetching of ads in bulk. For instance in Privad [49], each client subscribes to coarse-grained classes of ads the user is interested in and an anonymization proxy constantly fetches ads on the client's behalf. However none of these systems have attempted to look at the impact that prefetching ads has on the economics of the otherwise real-time nature of the online advertising industry. This chapter describes an ad architecture that is compatible with such systems and can help them become a feasible business model for privacy preservation.

Achieving an accurate prefetching model is in general hard, but the basic idea of content prefetching is relatively straightforward for applications such as web browsing [9, 82, 107]

and web search [70]. A naïve approach to achieving high energy savings is to prefetch as much content as possible in batches, perhaps once or twice a day (e.g., in the morning when the user wakes up). Prefetching can be managed by the client, independent of the server. Unfortunately, this approach is not practical for advertising systems, where ads are sold through on-demand auctions, and only if a client can immediately display an ad. By offering the client the option of downloading ads in batches, an advertising system actually undertakes a risk of revenue loss. At a high level, revenue loss can occur either because a prefetched ad is not shown within its deadline or is shown more times than required (this can happen if the ad is prefetched at the same time by multiple clients to ensure it will be shown eventually). On the other hand, the largest mobile advertising platforms are owned by the largest smartphone providers (e.g., Google's AdMob - Android, Apple's iAds - iPhone, Microsoft's MSN Ad Network - Windows Phone), thus there is an incentive to provide low-energy apps to attract users.

Today, ad platforms own the client access logs. However this data can potentially be brokered through a trusted entity that proxies traffic to the ad network on behalf of multiple users. It is possible that the client may fetch too few ads, thus achieving only limited energy savings. Or the client may fetch too many ads and not be able to display all of them by their deadlines, thus causing SLA violations for the ad infrastructure. The inaccuracy of the prediction model can be compensated for by scheduling ads in order of expiring deadlines and by introducing an overbooking model that probabilistically replicates ads across clients. Our evaluation shows that our approach is capable of reducing the energy consumed by an average client by 50%, while guaranteeing SLA violation rates of less than 3%.

5.7 Related work in prefetching web content

Ad systems have been the focus of several recent projects [11, 49, 138] mainly because of the privacy issues they raise. Web tracking [117] as well as information leaked by mobile apps [34, 36] are key to provide targeted ads. Privacy-preserving architectures such as Adnostic [138] and Privad [49] allow users to disclose only coarse-grained information about themselves, while personal profiles are kept local and use to rank relevant ads. These architectures build on the assumption that prefetching ads in bulk is a feature supported by ad servers. For instance, Privad uses subscription-based prefetching where the user manually subscribes to ads category of interest and corresponding ads are fetched at a proxy whenever available. The solution for ad prefetching presented in this chapter is both compatible with existing ad infrastructure as well as meets the requirements of the privacy-preserving ad systems discussed in this section.

Previous work have also looked at the energy cost of mobile ads. The authors of [64] found significant communication overhead when studying 13 popular Android apps and sketch the proposal for a middleware prefetching ads when network conditions are most favorable. While MobAds shares the same goal, one of the major focus points in the design and evaluation of MobAds is that it remain compatible with the current ad ecosystem.

Pathak et al. [110] also highlight the high network cost of advertising using a fine-grained energy profiler for smartphones. They tested the tool with 6 popular Android apps and found that ads in a game such as Angry Birds consume 45% of the total energy.

Section 5.9.4 discusses the high communication energy costs of mobile ads caused due to the 'tail time' of 3G networks. A long tail time improves responsiveness, but causes a large energy tail. Balasubramanian et al. [12] show that in 3G networks, about 60% of the energy consumed for a 50 kB download is tail energy. In a GSM network where the tail time is shorter network transfers consume 40% to 70% less energy compared to 3G, but suffer from higher network latency. WiFi is not affected by the tail energy problem, but incurs a high initialization cost for associating with an access point. Solutions that looked into determining the optimal tail time for different 3G networks [22,147], requires changes at the network operator's side.

MobAds builds on the basic principles of content prefetching. In particular, content prefetching has been studied extensively in the context of web browsing, both for desktops [62, 87, 101, 107] and mobile devices. For mobile devices, prefetching has been used to support disconnected operation [68, 148], or to reduce access latency and power consumption [9, 12, 148]. Predicting a user's web accesses typically relies on determining the probability of the user accessing web content based on previously accessed content, by extracting sequential and set patterns [57, 101], as well as temporal features (absolute and relative time of browsing) for higher prefetching accuracy [82]. MobAds's app usage prediction model is based only on temporal access patterns. The main difference between web content prefetching techniques and ad prefetching is that for web content the prefetching strategy is entirely client-driven, whereas for ads the server needs to be in control to minimize the risk of revenue loss.

In general, most of the existing literature on reducing tail energy propose solutions that require changes at the network operator side (e.g., adjusting the tail time [22, 147] or at the device's network stack (switching from low-power/low-bandwidth interfaces to high-power/high-bandwidth interfaces when network activity requires [111]).

MobAds explores an alternative approach: prefetching ads in bulk. Apart from the obvious privacy benefits for the end-client, prefetching can also amortize the tail energy cost among multiple ads. Experiments show that downloading 10 ads of size 1 kB (5 kB) in a bulk over AT&T 3G network consumes 8.6× (4.1× respectively) less energy than downloading them one every minute. Moreover, prefetching enables downloading ads at opportune times such as when the phone is being charged or WiFi connectivity is available. Finally, unlike solutions that require changes at the networking stack, a prefetching-based solution can be deployed on today's smartphones with minimal changes to existing ad infrastructures.

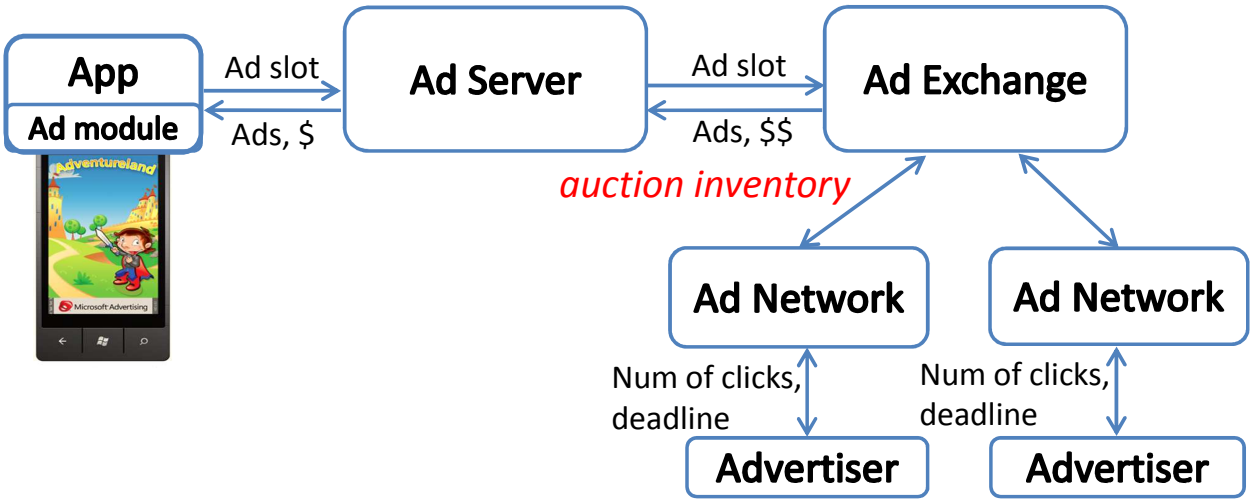


Figure 5.5: Architecture of a typical mobile ad system.

5.8 Feasibility and challenges in prefetching online advertisement

This section provides a description of how mobile advertising works today and discusses the challenges involved in supporting batch prefetching of ads with minimal changes to the current infrastructure.

5.8.1 Background on mobile advertising

As Figure 5.5 shows, in its most basic form, a mobile advertising system consists of five parties: mobile clients, advertisers, ad servers, ad exchanges, and ad networks. A mobile application includes an ad library module (e.g., AdControl for Windows Phones, AdMob for Android) which notifies the associated ad server any time an *ad slot* becomes available on the client's device. The ad server decides to monetize the ad slot by displaying an ad. Ads are collected from an *ad exchange*. Ad exchanges are neutral parties that aggregate ads from different third party ad networks and hold an auction every time a client's ad slot becomes available. The ad networks participating in the exchange estimate their expected revenue from showing an ad in such an ad slot and place a bid on behalf of their customers (i.e., the advertisers). An ad network attempts to maximize its revenue by choosing ads that are most appropriate given the context of the user to maximize the possibility of the user clicking on the ads. The ad network receives information about the user such as his profile, context, device type, from the ad server through ad exchange. Ad exchange runs the auction and chooses the winner with the highest bid.

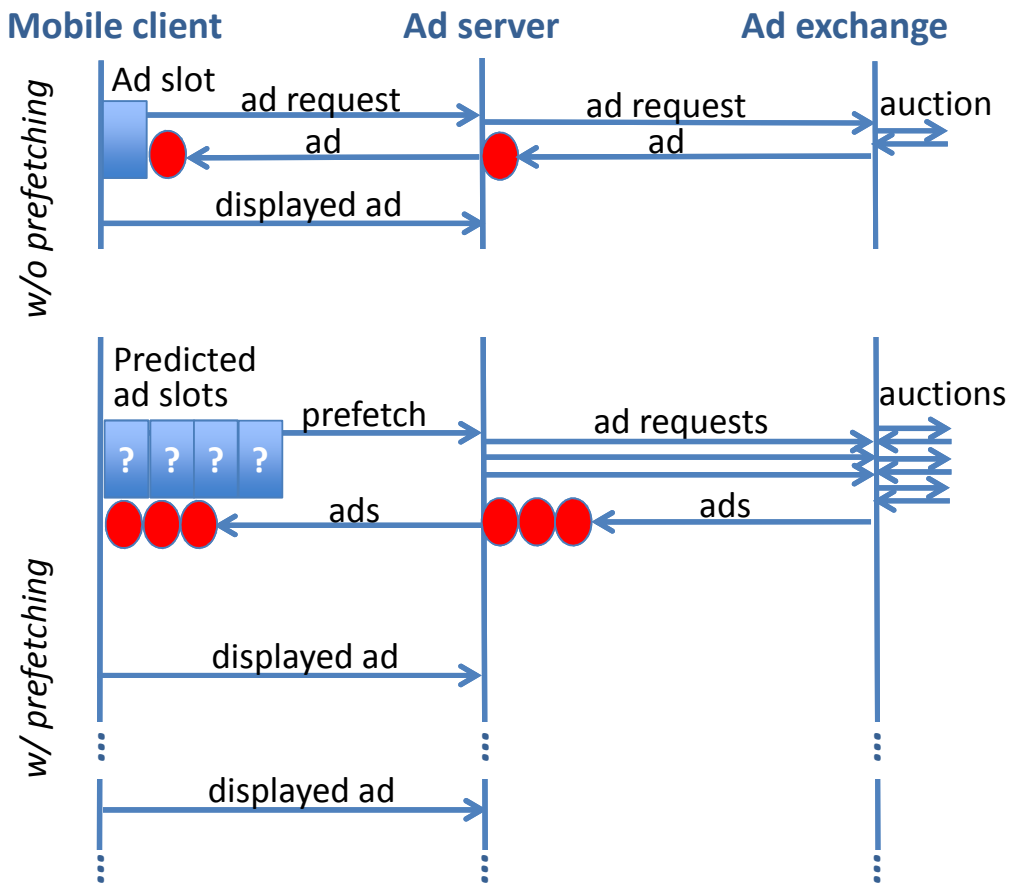


Figure 5.6: Ad system without and with ad prefetching (the proxy logic runs at the client and at the ad server).

Advertisers register with their ad networks by submitting an *ad campaign*. A campaign typically specifies an advertising budget and a target number of impressions/clicks within a certain deadline (e.g., 50,000 impressions delivered in 2 weeks). They can also specify a maximum cap on how many times a single client can see a specific ad and how to distribute ads over time (e.g., 150 impressions per hour).

The ad server is responsible for tracking which ads are displayed and clicked, and thus determining how much money an advertiser owes. The revenue of an ad slot can be measured in several ways, most often by views (Cost Per Impression) or click-through (Cost Per Click), the latter being most common in mobile systems. The ad server receives a premium on the sale of each ad slot, part of which is passed to the developer of the app where the ad was displayed.

5.8.2 A proxy-based ad prefetching system

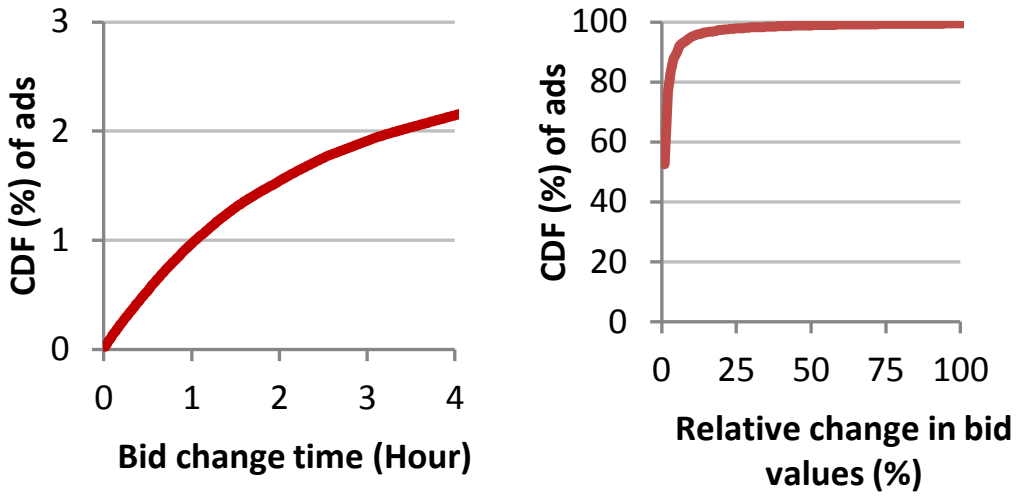
One way to incorporate ad prefetching into the existing ad ecosystem is to use a proxy between the ad server and the mobile client. A client with available ad slots contacts the proxy that prefetches a batch of ads from the ad exchange (through the ad server) and sends the batch to the client. After the client has displayed all ads of the batch, it contacts the proxy again and gets the next batch of ads. Such a solution is easy to implement in any existing smartphone app that receives ads through an ad control, a software module embedded within the app, provided by the ad server. The client-side of the prefetching logic can be implemented within the ad control, while the server-side logic can be implemented in the ad server, which acts as the prefetching proxy. Figure 5.6 shows how the ad system works today without ad prefetching, and how it can work with ad prefetching.

While it is easy to incorporate a prefetching proxy into the existing ad ecosystem, feasibility and advantage of prefetching depend on various factors. The two key decisions a prefetching model must take are 'what' to prefetch and 'when' to prefetch. These have been successfully addressed in the context of web browsing [9, 82, 107] and web search [70]. However, the key property that distinguishes ad prefetching from other prefetching scenarios is that *ads have deadlines and there are penalties if ads are prefetched but not served within their deadlines*. In fact, ad prefetching cannot be implemented as a stand-alone client action as for web browsing. A bad prefetching strategy that does not respect ad deadlines can adversely affect other parties involved in the ad ecosystem.

Ad deadlines. The deadline D of an ad may come from multiple sources. The advertiser typically wants all ads in an ad campaign to be served within a deadline. For example, an advertiser may start a campaign for an upcoming sale and the associated ads must be delivered before the sale ends. Even when an advertiser puts a long deadline, it expects the ad network to guarantee some SLA about the rate at which ads are served. For example, an advertiser may start a one-week ad campaign, but still want the guarantee that 100 impressions of its ads will be served per hour. In existing ad systems, these are the only factors affecting ads' deadlines since ads are delivered to clients within a short time period time (few hundred milliseconds) after being retrieved from the ad exchange.

With prefetching, however, other factors play into deciding an ad's deadline. Suppose the proxy serves ads within a serving period (smaller than the ad deadline specified by the advertiser). Since the bid price for the ads changes over time, the proxy (hence the ad server) takes some risks in prefetching ads. One type of risk is that it may not be able to serve all prefetched ads within the advertiser's deadline. Another type of risk is that the bid price for an ad may change within the serving period, and if the price actually goes up, the ad server could have made more revenue by collecting the ad from the ad exchange at a later point in time rather than at the beginning of the serving period.

To answer the question of estimating the risk involved, we sampled approximately 1 TB of log data from a production advertising platform spanning a 12 hour period in August 1, 2012. The data covers over several hundred million auctions and unique ads shown to users of a popular search engine across all search topics. The trace record for an auction lists all



(a) Bid change time (b) Relative price change

Figure 5.7: CDFs of (a) how often bid prices change for an ad and (b) relative price difference when a bid price changes.

the ads that participated in it (whether the ad was ultimately shown or not) and the bids corresponding to each ad. This trace is used to understand how often the bid price for an ad changes and by how much. It can be seen that most ads do not change their bid prices within the trace period. Figure 5.7(a) shows a portion of the CDF of the average time of an ad changing its bid price. As shown in the figure, less than 0.5% of the ads change their bid price within 30 minutes of their most recent price change (or their first appearance in the system). Figure 5.7(b) shows the CDF of the relative price change when an ad changes its bid: 95% of the bid changes are within 10% of the previous bid price.

The results above highlight that even though auction prices in the ad exchange are dynamic, the dynamics and hence the revenue risks of the ad server are relatively small for a small window of time. For example, if the ad server prefetches ads for a serving period of 30 minutes, the probability that any of the ads will change its bid price within the serving period is $< 0.5\%$, and even if an ad changes its bid price, the change will be $< 10\%$ in 95% of the cases. An ad server can choose a suitable value of serving deadline depending on the dynamics. Unless otherwise specified, we assume a serving deadline of 30 minutes in the rest of the chapter.

While prefetching, the actual deadline of an ad is the minimum of the deadline specified by the advertiser and the serving deadline the server uses for its prefetched ads. Deadlines specified by advertisers are typically longer than 30 minutes, and therefore, we consider the serving deadline as the deadline D for all prefetched ads.

5.8.3 Ad prefetching trade-offs

Ideally, a proxy would want to serve all prefetched ads within their deadline d. This is possible if the proxy can predict exactly how many ads it will be able to serve to its clients within a period of d. However, such predictions are unlikely to be accurate in practice. Hence, the ad infrastructure runs into the following two types of risks.

- **SLA violations:** SLA violations may happen if a proxy (hence the ad server) fails to deliver a prefetched ad within its deadline (e.g., an ad for a promotion is displayed after the promotion ends, or an ad is displayed when its bid price is significantly different from when it was prefetched).
- **Revenue loss:** Revenue earned by an ad server is related to the number of unique ads served to clients. A bad ad-prefetching and -serving system can cause revenue loss to the ad server: by serving ads after deadlines, by serving the same ad impression to more users than required, by not serving an ad even when a slot is available, etc.

There exists tradeoffs between the above two factors and network overhead. An aggressive strategy may prefetch more ads than can be served to the client within the ad deadline – this will minimize communication overhead but cause frequent SLA violations. On the other hand, a conservative prefetching approach may avoid SLA violations by prefetching a small number of ads, but will incur large communication overhead. The prefetching proxy may consider replicating ads to reduce SLA violation: it can send the same ad impression to multiple clients to maximize the probability that the ad impression will be served to at least one client. However, this may incur revenue loss. A good prefetching strategy needs to strike a good balance between these competing factors.

In the rest of the chapter, we will investigate these tradeoffs. In particular, we consider the following mechanisms, implemented in a proxy, and their effects on energy, SLA, and revenue.

- **App usage prediction:** The prefetching proxy estimates how many ads a client will be able to show in next T minutes (T is called the *prediction interval*) and prefetches that many ads from the ad server. A perfect prediction should result in an optimal communication energy, with no SLA violation and no revenue loss.
- **Overbooking:** The proxy may replicate one ad across multiple clients. In the presence of inaccurate predictions, this may reduce SLA violations, but increase revenue loss.

5.9 Energy cost of mobile ads

In this section, we empirically estimate how much energy mobile apps consume in order to communicate with ad servers? We do this with the top 15 ad-supported Windows Phone

apps. Note that some of the most popular apps (e.g., Facebook, YouTube) do not display any ads, while some other popular apps (such as Angry Birds) are ad-free paid apps; they are omitted from this study.

5.9.1 Communication costs for serving ads

Previous work highlighted significant overheads of ads in smartphone apps. In [64], the authors show that ads are responsible for non-negligible network traffic. Translating network traffic to communication energy, however, is nontrivial because communication energy depends on various other factors such as the current radio state, radio power model, radio bandwidth, etc. In [110], the authors propose *eprof*, a fine-grained energy profiler for smartphone apps that traces various system calls and uses power models to report energy consumption of various components of an app. Using this tool the authors demonstrate that third party modules consume significant energy on six apps they study (e.g., Flurry [2], a third party data aggregator and ad generator, consumes 45% energy in AngryBirds). The goal of the study was not to isolate the communication overhead of ad modules. In fact, with the *eprof*'s approach, accurately isolating communication overhead of a third party module alone is nontrivial when the app itself communicates with a backend server and communications of the app and third-party module interleave. This is because when multiple components share a radio for communication, it is not clear whom to attribute the nontrivial wake-up and tail energy of the radio.

5.9.2 Measurement methodology

In order to isolate the exact communication overhead of ads within an app, we use an approach different from *eprof* [110]. Given an app, we produce three versions of it: the original *ad-enabled* version, a modified *ad-disabled* version that does not communicate with the ad server and shows a locally cached ad, and a modified *ad-removed* version that does not show any ad. These versions are then executed with the same user interaction workload and measure their energy consumption. The difference between the first and the second version gives the communication energy overhead due to the ad module, and that between the second and the third version gives the non-communication energy overhead of the ad module. This approach is more accurate than *eprof* as we do not need to use any ad-hoc heuristics to distribute shared network costs between the app and ad modules. However, in taking this approach we need to address several challenges.

Measuring energy. To compare the ad module's communication energy with that of the app, we need to measure their communication energy as well as total energy. Thus, tools that give only total energy (such as a powermeter or battery level monitor) are not sufficient. We use WattsOn [96], a tool for estimating energy consumption of a Windows Phone app while running it on the Microsoft's Windows Phone Emulator (WPE). WPE can run any app binary downloaded from the Windows Phone Marketplace. When an app runs on WPE, WattsOn captures all network activities of the app and uses state-of-the-art power models

of WiFi, 3G radio, CPU, and display to estimate communication and total energy of the app. WattsOn also allows using various operating conditions (carrier, signal strength, screen brightness). Experiments show that WattsOn's estimation error is $< 5\%$ for networked apps, compared to the true energy measured by a power meter.

In these measurements, WattsOn was set to simulate the Samsung Focus phone with AT&T as the carrier. The phone uses 3G communication and enjoys 'good' signal strength and network quality, with average download and upload bandwidth of 2500 kbps and 1600 kbps respectively [135]. The display is configured with medium brightness.

Producing ad-disabled and ad-removed versions of an app. To produce an ad-disabled version of a given app, the communication between the ad modules and the ad servers must be disabled. This is relatively simple for most of the apps we tried: these apps include standard ad controls (such as Windows Ad Control or AdDuplex) that communicate with predefined ad servers. To disable such communication, DNS requests for their ad servers was redirected to the localhost interface (127.0.0.1). WattsOn ignores any communication redirected to this interface. Since the apps were run in an emulator, this was done easily by modifying the `/etc/hosts` file in the machine the WPE runs on. Most ad controls show a default locally-cached ad after failing to connect to ad servers, without affecting normal app operations.

The above simple trick does not work for apps (e.g., BubbleBursts) that dynamically download IP addresses of ad servers from the network. It also does not work for apps (e.g., Weather) that use the same backend server for downloading ads and useful application data. For such apps, binary rewriting techniques were used to modify app binaries to remove instructions that request new ads. Windows Phone apps are written in .Net, and the Common Compiler Infrastructure library [95] was used for binary rewriting.

Workload. We use each app three times and report the average energy consumption. In each run of an app, we use it for two minutes in its expected usage mode. For example, if the app is a game, we start it and play a few levels of the game; if the app is a news app, we open it, navigate to a few pages and read them.

A typical app's energy consumption depends on how it is used, e.g., which pages within the app are visited and for how long. For a fair comparison of ad-enabled, ad-disabled, and ad-removed versions of the same app, we run them with the same sequences of user interactions. More precisely, we record user interactions with the original app and replay the recorded interactions on all three versions of the app. Some apps show random behavior across runs. For example, the Bubble Burst game starts with a random game every time it is started. For such apps, we cannot replay a prerecorded sequence of user interactions, and hence we simply use all three versions independently for the same duration of time.

5.9.3 Energy overhead of in-app advertising

Figure 5.8 shows the energy measurements for the top 15 ad-supported Windows Phone apps. These apps use various ad controls such as Windows Phone Ad Control, AdMob,

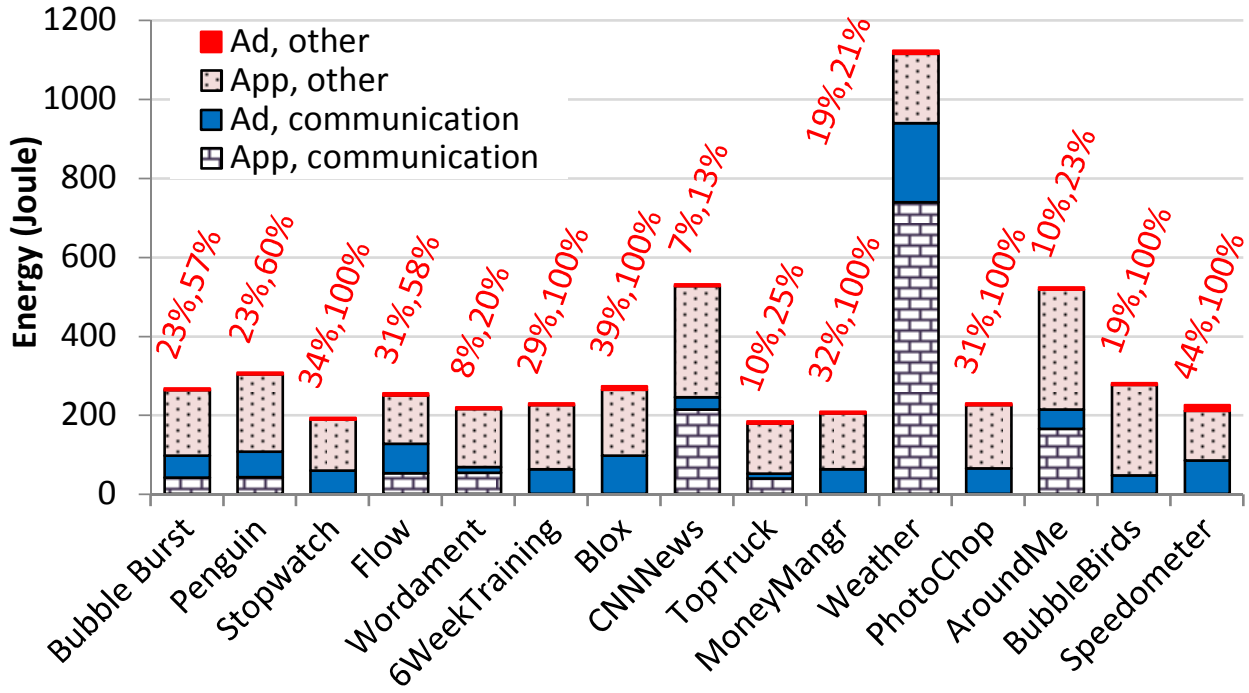


Figure 5.8: Energy consumed by top ad-supported WP apps. Both original and ad-disabled versions are run with the same sequence of user interactions for 2 minutes. The label $x\%, y\%$ on top of each bar means ads consume $x\%$ of total energy and $y\%$ of communication energy of the app. Ads consume significant communication energy, even for communication-heavy apps. *On average, ads consume* 23% *of total energy and* 65% *of communication energy.*

AdDuplex, Somaata, and DoubleClick, and some apps use multiple ad controls.

These measurements reveal several important points. First, ad modules consume a significant part of an app's energy. Across the 15 apps measured, ads are, on average, responsible for 23% of the total energy consumed by the app (including CPU, display, and communication), and 65% of the total communication energy. This overhead is significant, considering the fact that typical ads are small in size ($<$ 100 bytes for most textual ads). Second, the overhead of ads is bigger in apps such as BubbleBurst and StopWatch with no or small network activity. On the other hand, in apps such as CNNNews and Weather that need to communicate with the Internet, communication of an ad module can often piggyback on the phone's already-turned-on radio. Interestingly, overheads of ads are substantial even in communication-heavy apps such as Weather and CNNNews—without ads, these apps can keep the phone's radio in low power state more often. Third, most of the overhead of ad modules comes from communication: CPU and display constitute $<$ 8% of the total ad overhead.

5.9.4 Tail energy problem

Why do in-app ads consume more than half of the communication energy consumed by the app itself? Typically, in a GSM or 3G network the radio operates in three power states: 'idle' if there is no network activity, DCH (Dedicated Channel) in which a channel is reserved and high-throughput, low-delay transmission is guaranteed, and FACH (Forward Access Channel) in which a channel is shared with other devices and is used when there is little network traffic. The idle state consumes 1% of the power of the DCH state, and the FACH state consumes about half of the DCH power. After a transmission, instead of transitioning from the DCH to the idle state, the device spends some extra time in the DCH state and then in the FACH state—5 and 12 seconds respectively for an AT&T 3G network [96]. This delay, called 'tail time', determines how responsive the device is when new network activity starts. Each network provider decides on this trade-off: a longer tail time consumes more power but makes the device more responsive; a shorter tail time consumes less power but introduces delays [22].

Each time a user starts ad-supported mobile apps, ads are fetched one by one, and they are regularly refreshed during app operation. Downloading an ad takes no more than a few seconds, but once the ad's download has completed, the 3G connection stays open for the extra tail time. The energy consumed during this idle time, called 'tail energy', causes the ads' large energy overhead. Balasubramanian et al. [12] have shown that in a typical 3G network (with a tail time of 12.5 seconds), about 60% of the energy consumed for a 50 kB download is tail energy. The overhead is even bigger for shorter downloads, such as a typical ad of size 5 kB.

5.10 Ad prefetching with app usage prediction

The first challenge to address is to decide how many ads to prefetch and how often. Suppose, each ad comes with a deadline of D minutes and one ad is displayed every t minutes during app usage (t is also referred to as the size of an ad slot or the refresh period of an ad). For simplicity, let us assume for now that the client periodically prefetches ads once every T minutes (the prediction period). If the client could predict the number ad slots (k) available in the next round, it could prefetch exactly k ads, satisfying client's needs and without wasting any ads. This section explores whether such perfect prediction is possible in practice.

5.10.1 App usage prediction

The number of ad slots available in the future depends on how often the user is likely to use apps installed on his phone. We analyze two real user datasets to answer the following key questions:

1. Is app usage predictable based on users' past behavior?

2. What features of past app usage are useful in prediction?

Note that previous work considered predicting what apps will be used in a given context in order to preload apps [145] or to customize homescreen icons [128]. In contrast, MobAds's aim is to find *how long* apps will be used in a given time window.

Datasets. We use the following two datasets.

- Windows Phone logs: device logs of 1,693 WP users over roughly a month. Users were randomly selected worldwide, among a larger number of mobile users that opted into feedback. Logs report usage and performance measures such as battery life, bandwidth consumption and application usage.
- iPhone logs: device logs of 25 iPhone users [81,125]. Logs were collected by the LiveLab project at Rice University. The deployment involved 25 undergraduate iPhone users and lasted one year. Among others, logs report battery level and application usage.

The logs were filtered to eliminate apps which do not support ads, such as call application, SMS, alarm clock, and settings. We then assumed all remaining apps display an ad at startup time and refresh it every t minutes.

Predictability with past behavior. We first use information-theoretic measures to get insights into predictability of phone usage based on past behavior. Information entropy is a well-known metric that measures the level of uncertainty associated with a random process. It quantifies the information contained in a message, usually in bits/symbol. The entropy of a discrete random variable X is defined as

$$H(X) = \sum_{x \in X} p(x) \log_2 p(x)$$

where $p(x)$ is the probability mass function, $0 \leq p(x) \leq 1$. In our scenario, the variable X denotes the value of k in a given round of length T.

To understand how predictable X is in our datasets, we compute the entropy of the underlying process that determines the value of X. From a given dataset, we compute the PDF of X, with $\Pr(X = i)$ as the probability of having i ad slots in time T (i.e., the probability of the user using apps for i ad slots in a window of T minutes). Finally, we compute the entropy of X by using the above equation. For concreteness, assume that $T =$ 60 minutes, $t = 1$ minute. The value of k can be any integer within the range $[0, 60]$. Thus, the value $\log_2(61) \approx 6$ gives the upper bound of X's entropy.

Since entropy tells us about the uncertainty associated with a process, it can implicitly provide information about its predictability. When the entropy is 0, the outcome of the process is completely deterministic and hence completely predictable. On the other hand, when the process is completely random, $p(x)$ takes on a uniform distribution, and the corresponding upper bound on the entropy can be calculated using the above equation. In general, the lower the entropy, the lower is the information uncertainty associated with the process, and the easier it is to predict future outcomes based on history.

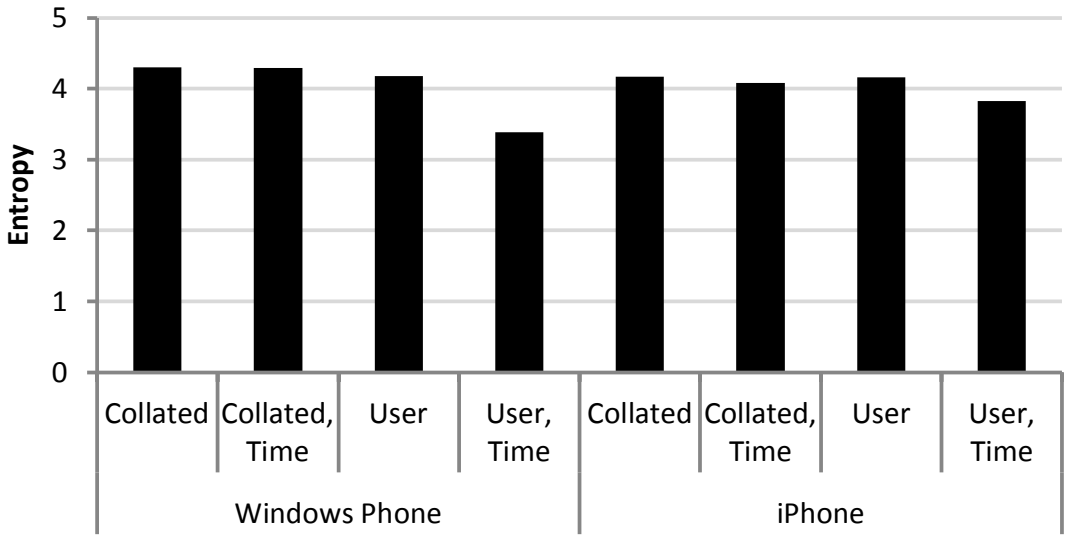


Figure 5.9: Entropy of app usage in two datasets, at different granularities.

To predict future outcomes of the value of k, past observations can be used at various granularities. Entropy at a given granularity will demonstrate how effective the granularity is in prediction. We consider two orthogonal dimensions to partition past observations:

1. *Collated vs. user-specific*: In a collated model, we assimilate traces of all users to form a collective trace. We then compute one entropy value of the collective trace. In a user-specific model, we consider each user trace in isolation, compute one entropy value for each user, and examine average entropy.

2. *Time independent vs. dependent*: In a time-independent model, we consider all rounds in the history alike and compute entropy from the PDF of $T = 1$ hour. In a time-dependent model, we maintain 24 PDFs—one for each hour of the day, compute their entropy values, and take the average entropy.[1]

The above two dimensions can produce four combinations of models.

Figure 5.9 shows the entropy of the two datasets under various models. The label `Collated,Time` on the x-axis, for example, denotes that we compute a time-dependent model over the collective trace of all users. The results highlight a number of key points. First, entropy is in general high. If we assume that past observations are completely random and hence useless in prediction, the entropy becomes $\log_2 61 \approx 6$. In both datasets and under all models, entropy is closer to this upper bound than the lower bound of 0. This suggests

[1]In a time-dependent model, one can consider partitioning a trace even further, such as one PDF for every hour of the week; however, with our limited dataset, such model becomes sparse and useless for prediction.

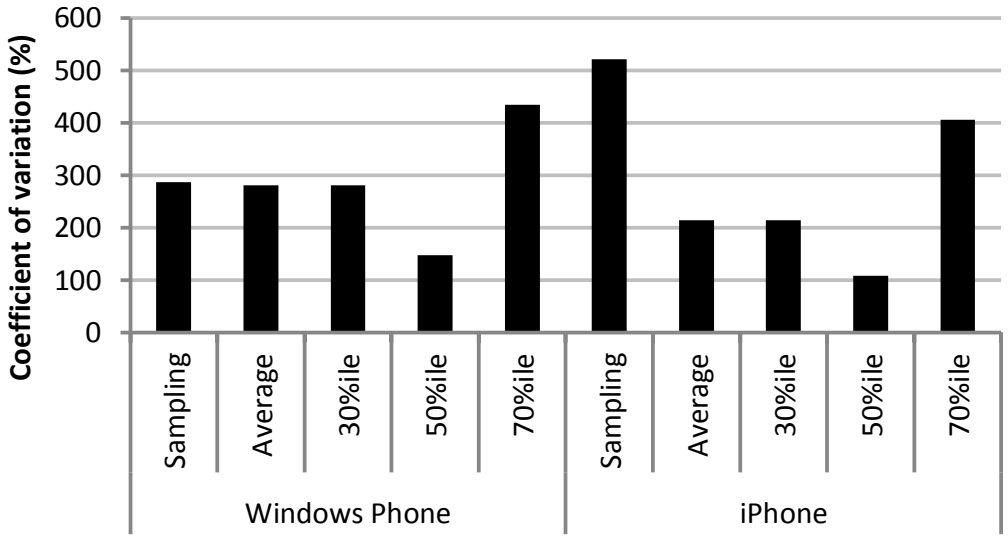


Figure 5.10: Coefficients of variation (RMSE/mean) of various predictors on user-specific, time-dependent models.

that app usage times in our dataset are mostly unpredictable. Note that this conclusion is based on the assumption that we use past app usage durations, user ID and time of use only. Prediction quality is likely to increase (and hence entropy is likely to decrease) if we use other information such as user's location context, as shown in previous work [145]. Furthermore, the ad server already has access to access logs that have temporal data, but might not have more intrusive data about the user for additional customization.

Figure 5.9 also shows that considering each user's trace in isolation makes the future outcomes of k a little bit more predictable (as shown by a reduction in entropy for `User`). Finally, considering data from different hours of the day separately further reduces the entropy. The entropy of the user-specific, time-dependent model is lower than all other models. In the rest of the chapter we consider each user's trace in isolation, and build one model for every hour of the day.

Prediction. We consider several statistical predictors to predict how many ad slots will be available in a given round. `Sampling` returns a random value sampled from the PDF of the user in the current hour of the day. `Avg` returns the average number of slots in the current hour of the day from past observations. `k'th percentile` returns the k'th percentile slot count in the current hour of the day from past observations. Figure 5.10 shows the result of using these predictors for our two datasets. We report coefficients of variations (root mean square error divided by mean) of the predictions. As shown, depending on k, the k'th percentile seems to be a good predictor for both the datasets and hence we use it in the rest of the chapter. In particular, for the coefficient of variation metric, 50th percentile performs

the best. This metric treats both underprediction and overprediction equally; in practice, they have different effects: underprediction forces the client to prefetch smaller numbers of ads, increasing its communication energy cost, while overprediction causes more frequent SLA violations. The best percentile to use depends on relative importance of energy and SLA violation (a detailed analysis is described below).

Note that above we assumed that the predictor uses only app usage history of users, in particular distribution of usage durations. It may be possible to improve prediction accuracy by using additional information such as user's context, correlation of usage patterns of various apps, etc. It can be expected that even though prediction can get better by using additional such information, as long as there are some prediction errors, ad prefetching will affect SLA and energy efficiency. One way for the proxy to limit the risk of causing SLA violations is to reduce the period of uncertainty on the status of the ads downloaded by the client.

5.10.2 Evaluating tradeoffs

This section experimentally evaluates the impact of an (imperfect) app usage predictor on energy-efficiency and number of SLA violations in a complete ad prefetching system. Our prefetching system works as follows. The client contacts the proxy when it has an ad slot but no ad to display. The proxy predicts how many ads (m) the client might need in the next T time, where T is the *prediction period*, smaller than the ad deadline D. It then collects m ads, by prefetching them from the ad exchange or from its pool of previously prefetched ads, and sends them to the client. If the client runs out of ads before time T, it similarly contacts the proxy again for additional ads. On the other hand, if the client has displayed only $m' < m$ ads during time T, it returns the undisplayed ads to the sever and gets a new batch of ads for the next prediction period. The undisplayed ads (that have now smaller lifetimes) are sent to other clients who have higher probabilities of showing ads.

For evaluating the above described system, we use the Windows Phone logs described in Section 5.10 to generate a realistic client workload. Our proxy implementation runs the previously described percentile predictor. Unless otherwise specified, we assume all ads have the same deadline D of 30 minutes and the same price. We also conservatively assume that a new ad is shown every 60 seconds while the user is using an app; a shorter ad refresh period will improve the relative benefit of prefetching on the battery lifetime. We use ads of size 5 kB each, which is the average ad size in the top 15 apps we used in Section 5.9. To measure energy, we capture network traces from our experiment, feed it into WattsOn used in Section 5.9, and measure communication energy for a phone using AT&T 3G wireless (the same setup we used in Section 5.9). We report the energy savings compared to a baseline client that fetches ads one by one, as in today's ad systems.

Impact of prediction periods. Increasing the prediction period should intuitively increase the energy efficiency since the client device fetches larger batches of ads. We first evaluate the impact of different prediction periods on the SLA violations, by using a 80th percentile

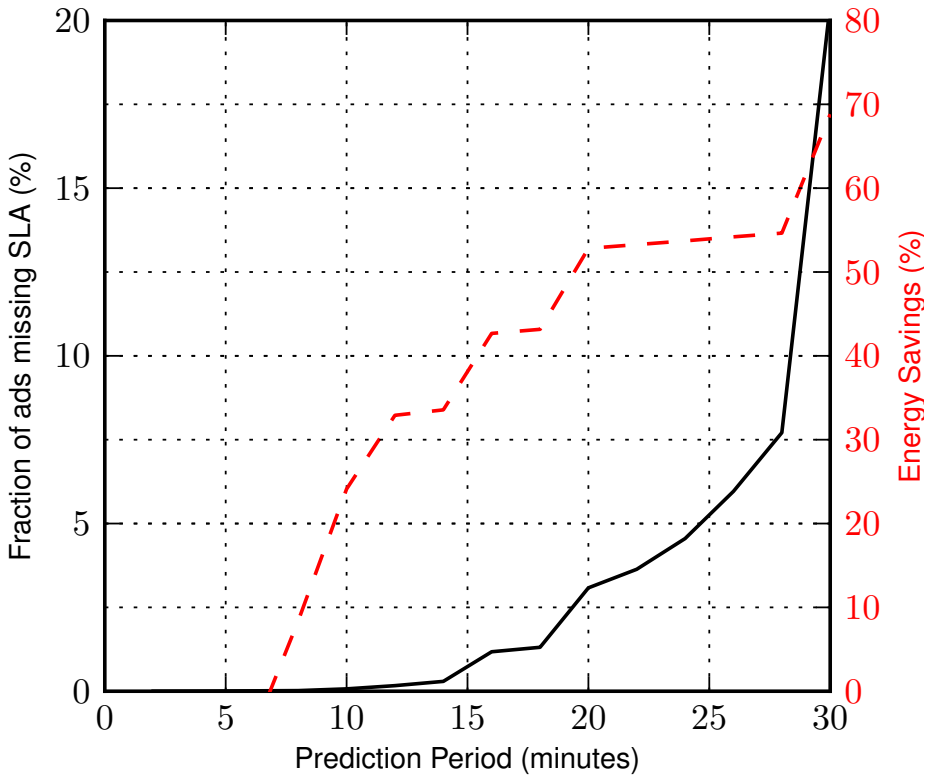


Figure 5.11: SLA violation rate and reduction in client's energy consumption for increasingly infrequent prediction. The number of ads prefetched is predicted using the 80th percentile prediction model.

prediction model. Figure 5.11 shows the percentage of the ad inventory that incurred an SLA violation and the corresponding reduction in energy consumption for increasingly longer prediction intervals. Since the ads have a deadline of 30 minutes, if the prediction period is longer than or equal to 30 minutes, then it is effectively equivalent to the client not reporting the status of the prefetched ads before their deadline. We see from the graph that for a prediction interval between 15–20 minutes the client achieves a net energy reduction of 40–50% while less than 3% of the ads in the inventory had an SLA violation. To achieve higher savings in energy consumption, a longer prediction interval can be chosen at the cost of increasing the SLA violations. We also observe that after 20 minutes the energy savings are relatively constant for increasing values of the prediction interval. For these reasons, for ads with deadline of 30 minutes, we consider a reasonable prediction interval to be 15 or 20 minutes.

Note that when there is network activity on the client (e.g., email syncing, Facebook updates, etc), some or all of the prefetching proxy's traffic can be delayed and piggybacked

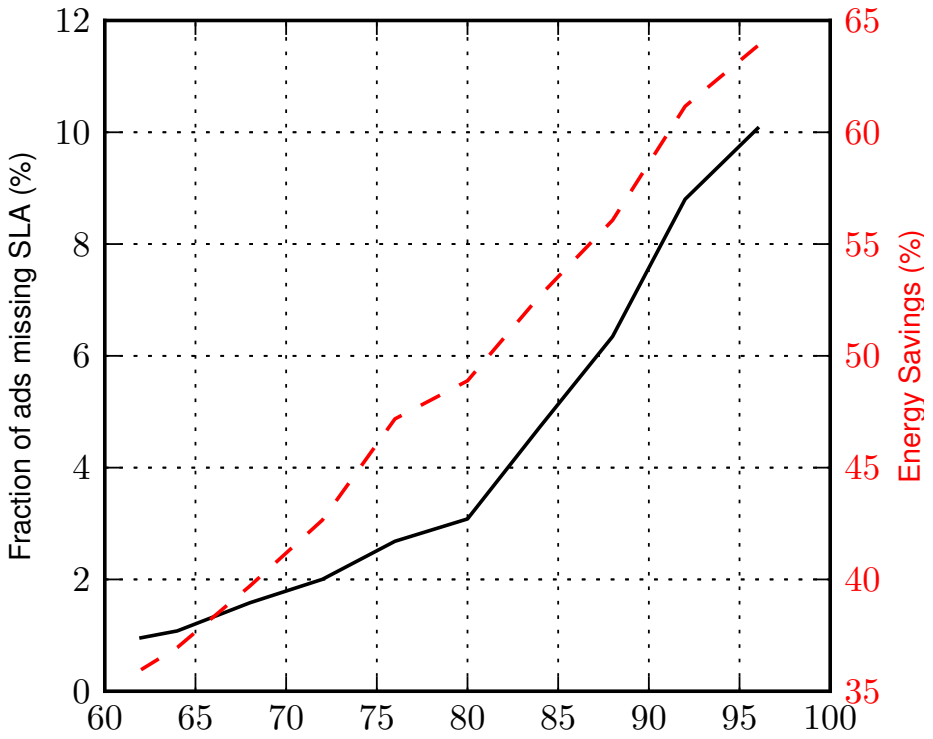


Figure 5.12: Trade-off between energy savings and SLA violations for increasingly larger prefetching rates (controlled by k of the kth percentile prediction model). The prediction interval is 15 minutes.

on top of other transfers. This reduces the impact of the radio's tail energy and therefore significantly improves the client's overall energy consumption. On the other hand, we cannot assume this to always be the case and thus report conservative energy savings numbers that assume no application network traffic in the background.

Prefetching rate. The next parameter that controls the energy savings is the prefetching rate or the k in the kth percentile model. The bigger k is, the larger the batches of ads prefetched by the client. We use a prediction interval of 15 minutes. We can see from Figure 5.12 that there is an almost linear increase in energy savings until it reaches the point where most of the batches are larger than the set of shown ads (around the 90th percentile). More interestingly, we see that the number of ads whose SLA is violated remains relatively low until the 80th percentile and then shoots up sharply.

Impact of ad deadlines. The final parameter to consider when trading off energy savings with the number of SLA-violated ads is the ad deadline. Figure 5.13 illustrates this trade-off using a 80th percentile predictor. Longer ad deadlines allow for less frequent prediction.

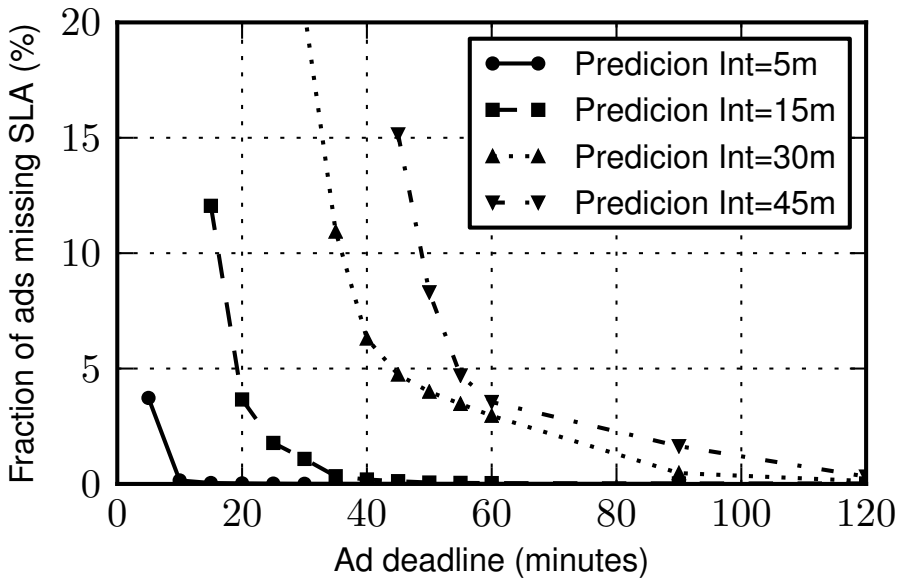


Figure 5.13: Tradeoff between ad deadlines and prediction period. The longer the ad deadlines, the smaller the client-proxy prediction period required for maintaining the same SLA violation rate. The prefetching is based on the 80th percentile prediction model.

For example, the same prediction period of 15 minutes that we considered above generates almost no SLA violations for ads with an hour or longer deadline. To put this in perspective, our analysis of an existing production ad platform (see Section 5.9) shows that only 1% of the ads change their bid price in less than an hour, which means that ad prefetching basically incurs no penalty on the ad infrastructure.

Overall, based on these experiments we conclude that for ad deadlines of 30 minutes, using the 80th percentile model with a prediction interval of 20 minutes reduces the energy overhead of ads by as much as 50% while impacting the SLAs of only 3% of the ad inventory. For ads with deadlines longer than 30 minutes, a prediction interval of 15 minutes is sufficient to eliminate the problem of SLA violations.

5.11 Overbooking model

The app usage prediction model guarantees a negligible number of SLA violations for ads with deadlines over 30 minutes, but could the proxy deal with shorter deadlines? As Figure 5.13 shows, SLA violations increase with shorter deadlines.

We explore whether advertising systems can take advantage of research in the area of overbooking of temporal resources [25, 131], which besides the traditional use cases of airline and hotel reservation systems, has been shown to be effective also for resource provisioning

in the cloud [139].

To support overbooking we modify our prefetching system as follows: (a) The proxy maintains a queue of unexpired, pending ads that have already been sent to some clients. (b) Each time a new client request is received, the proxy computes not only an estimation of how many ad slots the client will have in the next prediction interval, but also the probability of each of those slots being used. This can be computed from the PDF of historical slot counts of the user. (c) On a request of new ads, the proxy sends to the client not only a set of ads, but also the information about which ad to be shown in which slot. (d) The proxy can send new ads to a client, or *overbook* (or replicate) some of the pending ads.

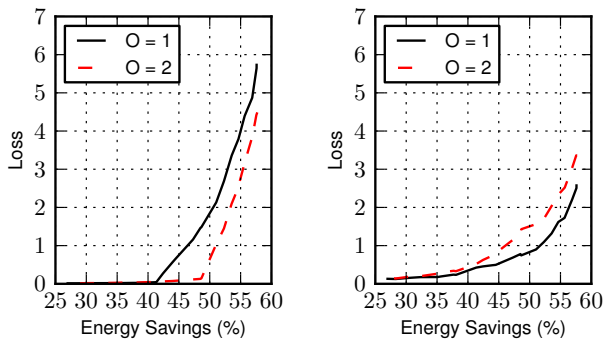
Intuitively, overbooking or sending an ad to multiple clients increases the chance that it will be displayed by at least one client and hence decreases the SLA violation rate. However, it entails the risk of displaying the same ad in multiple client slots while only being paid for one impression by the advertiser (i.e., revenue reduction may occur). The goal of the overbooking model is to maximize the number of distinct ads that can be shown given a certain number of client ad slots. In particular, prefetched ads which are unlikely to be shown, and only those, should be replicated across clients more aggressively. For the next experiments, we conservatively assume that there is no background network traffic for opportunistic notifications to the proxy.

Overbooking algorithm. Each time a client device requests a set of ads, the overbooking model attributes a *showing probability* to each of its pending ads. For a given pending ad, let S denote the set of ad slots (in different clients) it has been sent to, and let $P(S_i)$ denote the probability of the ith slot in S being used. Let X be the random variable denoting the number of times the pending ad will be displayed. Then,

$$
\begin{aligned}
P(X = 0) &= 1 - \prod_i (1 - P(S_i)) \\
P(X \geq 1) &= 1 - P(X = 0) \\
P(X = 1) &= P(S_1) \prod_{i \neq 1} (1 - P(S_i)) + \ldots \\
&\qquad P(S_n) \prod_{i \neq n} (1 - P(S_i)) \\
P(X > 1) &= P(X \geq 1) - P(X = 1)
\end{aligned}
$$

$P(X = 0)$ is the probability that an SLA miss will occur for the ad and $P(X \geq 1)$ is the probability that multiple displays will be made.

Each time a request is made for a batch of ads, the proxy iterates through the set of ads it has already retrieved from the ad exchange whose display status is unknown and verifies if the penalty for associating the ad with a given slot will increase or decrease. If the penalty decreases, the ad is associated to the slot that most minimizes its penalty. The penalty function is defined as:



(a) $\alpha = 0.1$ (b) $\alpha = 1$

Figure 5.14: Effect of different overbooking thresholds.

$$Penalty = P(X \geq 1) + P(X = 0) \times O$$

The parameter O is the *overbooking threshold* value that the proxy uses to tune the aggressiveness of the overbooking model. The smaller the value of O, the more aggressive overbooking is. We use a fixed value of $O = 1$ for all ads, but it could also be used to prioritize certain types of ads over others, potentially based on revenue.

The above penalty function can be computed for ads that have already been sent to other clients (and hence the set S of slots they are attached to is non empty). For an ad which is currently not sent to any client, however, this is not true. For such an ad, we use the following procedure to pick a slot. Each such ad has a lifetime d, computed as the difference between its original deadline D and the time elapsed since it was first prefetched. We would like to put shortly expiring ads into first few slots, with higher probability of being used. We therefore assign the ad to any of the first $d/D \times B$ slots, where B is the batch size predicted. For example, if an ad is not attached to any slot (in any client) yet, but its lifetime is only 1/3rd of its deadline, we assign it to any of the first 1/3rd slots. This ensures that the shorter the lifetime, the more aggressively the proxy puts the ad on slots with higher showing probability.

Evaluating overbooking. Overbooking results in decreased SLA violations at the cost of revenue loss. Different ad networks may have different preferences towards these two factors. For example, in existing ad systems, when the ad proxy retrieves an ad from the ad exchange, the ad network who won the auction assumes that it will be immediately displayed on the client. This is important because the ad network has to manage its own campaigns from different advertisers. Thus, it is expected that the proxy will try to reduce the SLA violations as much as possible, even at the cost of losing some revenue (when ads are displayed multiple times). This preference can be controlled by the overbooking threshold O.

Figure 5.14 shows the effects of two different overbooking thresholds. We unify SLA

violation and revenue loss into a single loss metric: $(\alpha \times sla\ violations + (1-\alpha) \times rev\ loss)$, which allows ad networks to weigh the two factors according to their preference. We report the loss metric for two values of α. As shown, when SLA violation is more important than revenue loss (i.e., $\alpha = 1$), the ad network should use aggressive overbooking, with a smaller value of the overbooking threshold (e.g., $O = 1$). On the other hand, when revenue is more important, the ad network should use conservative overbooking, with a larger value of O.

5.12 Summary

Bubbles provides a new way of thinking about managing data on client devices. We can exploit the contextual nature of human interaction along with the rich information database afforded by mobile smartphones to better manage the sharing of information. Bubbles makes the following technical contributions:

1. **User abstractions:** It extends the Application-Storage-Template design pattern to the client devices to introduce the "bubble" and "foam" user abstractions.

2. **Context-specific isolation:** The "bubble" and "foam" abstractions provide a user-intuitive mechanism to provide context-specific data isolation on mobile devices.

3. **Prototype implementation:** A prototype type implementation of Bubbles was built and evaluated on top of the Android operation system.

For users who do not want to actively place their sensitive information on the online host or for service providers who do not want to take on additional liability by holding the user's personal information, we need to provide the benefits of connected services without sacrificing privacy. At it's extreme case the advertising system which is a very real-time oriented system provides a challenge to disconnected activity. MobAds evaluates the feasibility of solving this problem with algorithms that ensure that the online service provider does not face monetary losses while the user still enjoys additional privacy. MobAds makes the following technical contributions:

1. **Energy overhead of mobile ads:** It proposes a methodology for accurately measuring the energy overhead of mobile ads and gives an estimate of such an overhead based on popular Windows Phone apps

2. **Prediction models:** We study the predictability of app usage behavior and derive personalized, time-based models based on roughly 1,700 iPhone and Windows Phone user traces

3. **Prefetching and overbooking:** We model the problem of ad prefetching as an overbooking problem where the ad server can explicitly tune the risk of SLA violations and revenue loss

4. **Evaluation of feasibility:** We evaluate the feasibility of the proposed approach and quantifies the energy savings for a realistic population of mobile users.

Chapter 6

Conclusion

This dissertation put forth a software development architecture for online services to provide soft "privacy guarantees" – a technologically enforced mechanism to ensure that the user's data is always used within the realm of the privacy policies specified by the user.

Online services are complex applications that include a number of different functionalities. One of the primary motivations for maintaining a cloud service is the ability to aggregate data from different users to create learning models. In order to ensure that these services do not either inadvertently or maliciously expose information about specific individuals we propose Gupt, a differentially private data analysis system. Gupt is a generic system that takes arbitrary binaries and ensures that the output of the program is always differentially private.

Other parts of these online applications include the business and presentation logic that stores and transforms user data to a form that appeals to the user. Many of these applications are buggy and mis-configured if not malicious itself. To circumvent this risk, we propose the use of Rubicon in the trusted code base. Rubicon allows users to specify their privacy policies as access control lists which are extremely simple to use. In the background, Rubicon transparently converts these ACLs into information flow control rules. As long as the application conform to the Application-Storage-Template design pattern (an extension to the multi-tired architecture), the user will not be deprived of any functionalities.

Privacy of the user's data cannot be guaranteed without safeguarding the client device. Bubbles is a system that extends the notion of Application-Storage-Template to client devices and adapts the system to context-centric privacy control. The data captured by the device is held within a context attached bubble and the user is free to share (explicitly) data between the bubbles and is always made aware of the privacy risks involved. Additionally, in spite of these restrictions, a number of users would still be unwilling to send their data to the cloud. We use the online ad system as a canonical example of an internet connected service which exposes the user to a number of privacy risks. A number of papers have been published that provide solutions for user data privacy but require that these ads be prefetched to the client device (or a proxy). This affects the real-time nature of the advertising system. This dissertation expands on this to provide algorithms that allow ad prefetching

to be performed without affecting the revenues of the advertising systems.

Finally, the privacy of a user is not strictly guaranteed even if all of these solutions are used, since a number of side channels including timing channels, electromagnetic leaks, power consumption, sound or even the human brain still exist. The identification of these side channels and protecting against them is not dealt with in this dissertation and is a field gaining crucial importance.

The systems presented in this dissertation work in tandem with each other and complement each other to complete the privacy protection story. The use of these concepts will vastly increase the cost of a privacy breach.

Bibliography

[1] Android security overview. `http://source.android.com/tech/security/`.

[2] Flurry. `http://www.flurry.com`.

[3] Webware 100 winners! `http://www.cnet.com/100/`.

[4] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[5] ABI Research. Android overtakes apple with 44% worldwide share of mobile app downloads. `http://goo.gl/gULCw`.

[6] N. Anciaux, L. Bouganim, H. H. van, P. Pucheral, and P. M. Apers. Data degradation: Making private data less sensitive over time. In *CIKM*, 2008.

[7] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *SOSP*, 2011.

[8] App Brain. Free vs. paid android apps. `http://www.appbrain.com/stats/android-app-downloads`.

[9] T. Armstrong, O. Trescases, C. Amza, and E. de Lara. Efficient and transparent dynamic content updates for mobile clients. In *Proc. of MobiSys '06*, pages 56–68. ACM, 2006.

[10] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, 2008.

[11] M. Backes, A. Kate, M. Maffei, and K. Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *Proc. of IEEE S&P*, 2012.

[12] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proc. of IMC*, 2009.

[13] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD*, 1986.

[14] M. Barbaro and T. Zeller. A face is exposed for aol searcher no. 4417749. *The New York Times*, Aug. 2006.

[15] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. *Proc 10*, 1(MTR-2997):118121, 1976.

[16] D. Booth and C. K. Liu. Web services description language (wsdl) version 2.0 part 0: Primer. *W3C Recommendations*, June 2007.

[17] K. Borders, V. Eric, E. Weele, B. Lau, and A. Prakash. Protecting Confidential Data on Personal Computers with Storage Capsules. In *USENIX Security*, 2008.

[18] M. Brown. *BeOS: porting UNIX applications*. Morgan Kaufmann, 1998.

[19] J. A. Calandrino, A. Kilzer, A. Narayanan, E. W. Felten, and V. Shmatikov. "you might also like: " privacy risks of collaborative filtering. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 231–246, Washington, DC, USA, 2011. IEEE Computer Society.

[20] R. Chandra, P. Gupta, and N. Zeldovich. Separating Web Applications from User Data Storage with BStore. In *WebApps*, 2010.

[21] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A software-hardware architecture for self-protecting data. In *ACM Conference on Computer and Communications Security*, pages 14–27, 2012.

[22] M. Chuah, W. Luo, and X. Zhang. Impacts of inactivity timer values on UMTS system capacity. In *Proc. of WCNC '02)*, 2002.

[23] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *ISCA*, pages 482–493, New York, NY, USA, 2007. ACM.

[24] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In M. Burmester, G. Tsudik, S. Magliveras, and I. Ilic, editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin / Heidelberg, 2011.

[25] P. Davis. Airline ties profitability to yield management. *SIAM News*, 1994.

[26] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Operating Systems Design and Implementation*, October 2004.

[27] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[28] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, July 1977.

[29] C. Dwork. Differential Privacy. In *ICALP*, 2006.

[30] C. Dwork and J. Lei. Differential privacy and robust statistics. In *STOC*, 2009.

[31] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006.

[32] P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. In *EuroSys*, pages 301–313, New York, NY, USA, 2008. ACM.

[33] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *SOSP*. ACM, 2005.

[34] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proc. of NDSS '11*, February 2011.

[35] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth.

TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[36] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 393–407. USENIX Association, October 2010.

[37] A. Ermolinskiy, S. Katti, S. Shenker, L. Fowler, and M. McCauley. Towards Practical Taint Tracking. Technical Report UCB/EECS-2010-92, UC Berkeley, June 2010.

[38] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[39] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *WebApps*, 2011.

[40] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[41] M. Fredrikson and B. Livshits. Repriv: Re-envisioning in-browser privacy. In *IEEE Symposium on Security and Privacy*, 2011.

[42] S. R. Ganta, S. P. Kasiviswanathan, and A. Smith. Composition attacks and auxiliary information in data privacy. In *KDD*, 2008.

[43] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

[44] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, pages 129–148, 2011.

[45] D. B. Gifn, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[46] E. Goffman. *The presentation of self in everyday life*. Garden City, NY: Doubleday Anchor Books, 1959.

[47] Google Inc. Google admob ads sdk. `https://developers.google.com/mobile-ads-sdk/docs/admob/intermediate`.

[48] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA*, 2011.

[49] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, 2011.

[50] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving ads from localhost for performance, privacy, and profit. In *HotNets*, 2009.

[51] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *USENIX Security*, 2011.

[52] W. R. Harris, S. Jha, and T. Reps. Difc programs by automatic instrumentation. In *ACM CCS*, pages 284–296, New York, NY, USA, 2010. ACM.

[53] M. Hay, V. Rastogi, G. Miklau, and D. Suciu. Boosting the accuracy of differentially

private histograms through consistency. *Proc. VLDB Endow.*, 3:1021–1032, September 2010.

[54] A. Holovaty, J. Kaplan-Moss, A. Holovaty, and J. Kaplan-Moss. The django template system. In *The Definitive Guide to Django*, pages 31–58. Apress, 2008.

[55] IAB Europe and IHS Screen Digest. Global mobile advertising market valued at $5.3 billion in 2011. `http://goo.gl/pyoTS`.

[56] ITRS Working Group. International technology roadmap for semiconductors 2009 report. Technical report, 2009.

[57] Z. Jiang and L. Kleinrock. Web prefetching in a mobile environment. *IEEE Personal Communications*, 5(5), 1998.

[58] M. Johnson and F. Stajano. Implementing a multi-hat pda. In *Security Protocols*, pages 295–307. Springer, 2007.

[59] A. Juels. Targeted advertising ... and privacy too. In *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA*, CT-RSA 2001, pages 408–424, London, UK, UK, 2001. Springer-Verlag.

[60] J. Kannan, P. Maniatis, and B.-G. Chun. Secure data preservers for web services. In *WebApps*, 2011.

[61] V. Karwa, S. Raskhodnikova, A. Smith, and G. Yaroslavtsev. Private analysis of graph structure. In *VLDB*, 2011.

[62] F. Khalil, J. Li, and H. Wang. Integrating recommendation models for improved web page prediction accuracy. In *Australasian Conference on Computer Science*, 2008.

[63] A. Khan, V. Subbaraju, A. Misra, and S. Seshan. Mitigating the true cost of advertisement-supported free mobile applications. In *HotMobile*, 2012.

[64] A. J. Khan, V. Subbaraju, A. Misra, and S. Seshan. Mitigating the true cost of advertisement-supported "free" mobile applications. In *Proc. of the HotMobile '12*, 2012.

[65] D. Kifer. Attacks on privacy and definetti's theorem. In *SIGMOD*, 2009.

[66] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. volume 5352 of *Lecture Notes in Computer Science*. Springer, 2008.

[67] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *USENIX Security*, 2002.

[68] A. Komninos and M. Dunlop. A calendar based internet content pre-caching agent for small computing devices. *J. of Personal and Ubiquitous Computing*, 12(7), 2008.

[69] W. Koo, T. Mew, G. Kwan, J. Lee, C. Li, and R. Quan. Friendshare: A decentralized, consisten storage repository for collaborative file sharing, 2008.

[70] E. Koukoumidis, D. Lymberopoulos, K. Strauss, J. Liu, and D. Burger. Pocket cloudlets. In *ASPLOS*, 2011.

[71] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *SOSP*, pages 321–334, New York, NY, USA, 2007. ACM.

[72] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *HotNets*. SIGCOMM, 2007.

[73] O. Laadan and S. Hallyn. Linux-cr: Transparent application checkpoint-restart in linux. In *Proceedings of the Ottawa Linux Symposium*, 2010.

[74] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *USENIX ATC*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[75] H. A. Lagar-Cavilla, J. A. Whitney, R. Bryant, P. Patchin, M. Brudno, E. de Lara, S. M. Rumble, M. Satyanarayanan, and A. Scannell. Snowflock: Virtual machine cloning as a first-class cloud primitive. *ACM Trans. Comput. Syst.*, 29(1):2:1–2:45, Feb. 2011.

[76] K. Lauter, M. Naehrig, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, 2011.

[77] Y. Lee, S. S. Iyengar, C. Min, Y. Ju, S. Kang, T. Park, J. Lee, Y. Rhee, and J. Song. Mobicon: a mobile context-monitoring platform. *Commun. ACM*, 55(3):54–65, Mar. 2012.

[78] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *Proc. of HotMobile '12*, 2012.

[79] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor. Optimizing linear counting queries under differential privacy. In *PODS*, 2010.

[80] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP*, pages 321–334, 2009.

[81] LiveLab traces. Rice university. `http://livelab.recg.rice.edu/traces.html`.

[82] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: instant web browsing for mobile devices. In *ASPLOS*, pages 1–12. ACM, 2012.

[83] J. MacDonald. *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[84] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006.

[85] A. Machanavajjhala, A. Korolova, and A. D. Sarma. Personalized social recommendations: accurate or private. *Proc. VLDB Endow.*, 4(7):440–450, Apr. 2011.

[86] P. Maniatis, D. Akhawe, K. Fall, E. Shi, and D. Song. Do you know where your data are? secure data capsules for deployable data protection. In *HotOS*, 2011.

[87] E. P. Markatos and C. E. Chronaki. A top-10 approach to prefetching on the web. In *Proc. of INET*, 1998.

[88] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *IEEE Security and Privacy*, pages 143–158, 2010.

[89] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, 2008.

[90] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.

[91] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *FOCS*,

2007.

[92] P. B. Menage. Adding Generic Process Containers to the Linux Kernel. In *Linux Symposium*. Google Inc., June 2007.

[93] R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[94] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *Trans. Storage*, 7(4):14:1–14:20, Feb. 2012.

[95] Microsoft Research. Common Compiler Infrastructure (CCI). `http://research.microsoft.com/en-us/projects/cci/`.

[96] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *Procs of MOBICOM '12*. ACM, August 2012.

[97] P. Mohan, S. Nath, and O. Riva. Prefetching mobile ads: Can advertising systems afford it? In *Proceedings of the 8th ACM european conference on Computer Systems*, EuroSys '13, New York, NY, USA, 2013. ACM.

[98] P. Mohan, A. G. Thakurta, E. Shi, D. Song, and D. E. Culler. GUPT: Privacy preserving data analysis made easy. In *Proceedings of the 38th SIGMOD international conference on Management of data*, SIGMOD '12, New York, NY, USA, 2012. ACM.

[99] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, New York, NY, USA, October 1997. ACM.

[100] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[101] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Trans. on Knowledge and Data Engineering*, 2003.

[102] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, 2008.

[103] J. Newsome and D. Song. Dynamic taint analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[104] H. Nissenbaum. Privacy in context: Technology, policy, and the integrity of social life. 2009.

[105] K. Nissim, S. Raskhodnikova, and A. Smith. Smooth sensitivity and sampling in private data analysis. In *STOC*, 2007.

[106] OWASP: The open web application security project. Top 10 2010, 2010.

[107] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[108] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[109] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. CLAMP: Practical Prevention of Large-Scale Data Leaks. In *IEEE Security and Privacy*, 2009.

[110] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof. In *Proceedings of the 7th ACM European conference on Computer Systems (Eurosys)*, 2012.

[111] T. Pering, Y. Agarwal, R. Gupta, and R. Want. CoolSpots: reducing the power consumption of wireless mobile devices with multiple radio interfaces. In *Proc. of*

MobiSys, 2006.

[112] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP*, pages 85–100, 2011.

[113] R. Pressman and D. Ince. *Software engineering: a practitioner's approach*, volume 5. McGraw-hill New York, 1992.

[114] V. Rastogi and S. Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD*, 2010.

[115] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *IEEE S&P*, 2012.

[116] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. *Security and Privacy, IEEE Symposium on*, pages 224–238, 2012.

[117] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, April 2012.

[118] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *PLDI*, pages 63–74, New York, NY, USA, 2009. ACM.

[119] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: security and privacy for mapreduce. In *NSDI*, 2010.

[120] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE JSAC*, 21, 2003.

[121] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security*, 2004.

[122] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: a new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, 2012.

[123] J. Seifert, A. De Luca, B. Conradi, and H. Hussmann. Treasurephone: Context-sensitive user data protection on mobile phones. *Pervasive Computing*, pages 130–137, 2010.

[124] A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In *PET*, 2002.

[125] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: measuring wireless networks and smartphone users in the field. In *ACM SIGMETRICS Perform. Eval. Rev*, volume 38. ACM, December 2010.

[126] M. Sherr and M. Blaze. Application containers without virtual machines. In *VMSec*, pages 39–42, New York, NY, USA, 2009. ACM.

[127] E. Shi, A. Perrig, and L. van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *IEEE Security and Privacy*, pages 154–168, 2005.

[128] C. Shin, J.-H. Hong, and A. Dey. Understanding and Prediction of Mobile Application Usage for Smart Phones. In *Ubicomp*, 2012.

[129] K. Singh, S. Bhola, and W. Lee. xBook: Redesigning Privacy Control in Social Networking Platforms. In *USENIX Security*, 2009.

[130] A. Smith. Privacy-preserving statistical estimation with optimal convergence rates. In *STOC*, 2011.

[131] B. Smith, J. Leimkuhler., and R. Darrow. Yield management at american airlines. *Interfaces*, 22(1):8–31, 1992.

[132] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 85–96, New York, NY, USA, 2004. ACM.

[133] J. Sukowaty. Google cans snooping employee... again, 2010. `http://www.toptechreviews.net/tech-news/google-cans-snooping-employee-again/`.

[134] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 2002.

[135] W. L. Tan, F. Lam, and W. C. Lau. An empirical study on the capacity and performance of 3g networks. *IEEE Transactions on Mobile Computing*, 7:737–750, June 2008.

[136] M. Tiwari, P. Mohan, A. Osheroff, H. Alkaff, E. Shi, E. Love, D. Song, and K. Asanovi. Context-centric security. In *Proceedings of the 7th USENIX conference on Hot topics in security*, HotSec'12, 2012.

[137] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *17th Network and Distributed System Security Symposium*, 2010.

[138] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy Preserving Targeted Advertising. In *Proc. of NDSS '10*, 2010.

[139] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *Proc. of OSDI '02*, Dec. 2002.

[140] H. H. van, M. Fokkinga, and N. Anciaux. A framework to balance privacy and data usability using data degradation. In *CSE*, 2009.

[141] A. F. Westin and L. Blom-Cooper. *Privacy and freedom*, volume 67. Atheneum New York, 1970.

[142] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security*, 2002.

[143] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Computer Science Department, Stony Brook University, October 2004. `www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf`.

[144] X. Xiao, G. Bender, M. Hay, and J. Gehrke. ireduct: differential privacy with reduced relative errors. In *SIGMOD*, 2011.

[145] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast app launching for mobile devices using predictive user context. In *Proc. of MobiSys*, pages 113–126, 2012.

[146] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula,

and N. Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, January 2010.

[147] J.-H. Yeh, J.-C. Chen, and C.-C. Lee. Comparative Analysis of Energy-Saving Techniques in 3GPP and 3GPP2 Systems. *IEEE Transactions on Vehicular Technology*, 2009.

[148] L. Yin and G. Cao. Adaptive power-aware prefetch in wireless networks. *IEEE Trans. on Wireless Comms*, 2004.

[149] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*. ACM, 2009.

[150] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, Aug. 2002.

[151] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI*, Berkeley, CA, USA, 2006. USENIX Association.

[152] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI*, 2006.

[153] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System Support for Derived Data Management. In *VEE*, pages 63–74, New York, NY, USA, 2010. ACM.