Personal Data Management in the *Internet of Things*

by

Rayman Preet Singh

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Due to a sharp decrease in hardware costs and shrinking form factors, networked sensors have become ubiquitous. Today, a variety of sensors are embedded into smartphones, tablets, and personal wearable devices, and are commonly installed in homes and buildings. Sensors are used to collect data about people in their proximity, referred to as *users*. The collection of such networked sensors is commonly referred to as the *Internet of Things*. Although sensor data enables a wide range of applications from security, to efficiency, to healthcare, this data can be used to reveal unwarranted private information about users. Thus it is imperative to preserve data privacy while providing users with a wide variety of applications to process their personal data.

Unfortunately, most existing systems do not meet these goals. Users are either forced to release their data to third parties, such as application developers, thus giving up data privacy in exchange for using data-driven applications, or are limited to using a fixed set of applications, such as those provided by the sensor manufacturer. To avoid this tradeoff, users may chose to host their data and applications on their personal devices, but this requires them to maintain data backups and ensure application performance. What is needed, therefore, is a system that gives users flexibility in their choice of data-driven applications while preserving their data privacy, without burdening users with the need to backup their data and providing computational resources for their applications.

We propose a software architecture that leverages a user's personal virtual execution environment (VEE) to host data-driven applications. This dissertation describes key software techniques and mechanisms that are necessary to enable this architecture. First, we provide a proof-of-concept implementation of our proposed architecture and demonstrate a privacy-preserving ecosystem of applications that process users' energy data as a case study. Second, we present a data management system (called *Bolt*) that provides applications with efficient storage and retrieval of time-series data, and guarantees the confidentiality and integrity of stored data. We then present a methodology to provision large numbers of personal VEEs on a single physical machine, and demonstrate its use with LinuX Containers (LXC). We conclude by outlining the design of an abstract framework to allow users to balance data privacy and application utility.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

In recent years, networked sensors have shown unprecedented growth in adoption, primarily driven by their decreasing cost and form factor. For instance, homes and buildings are increasingly outfitted with sensors that measure and report temperature, water and energy consumption, or detect motion, water leaks, open doors and windows [3, 120]. It is estimated that 90 million homes in the US will be equipped with sensors by 2017 [1, 40]. Similarly, smart meters that measure energy consumption have also been widely deployed. Approximately 36 million homes in the US [42] (and 4 million homes in Ontario, Canada [41]) have smart meters installed. Wearable sensors such as Fitbit [14], that monitor users' health are also being increasingly adopted [99]. Smartphones and tablets have evolved into multi-sensor devices and have various embedded sensors, such as, GPS, accelerometers, gyroscopes, and heart-rate monitors. Analysts predict that by 2020, 50 billion networked sensors will be deployed across our living environments [34], collectively dubbed as the *Internet of Things (IoT)*.

The rapid growth of networked sensors has enabled a wide variety of applications. For instance, energy data analytics applications use data from smart meters and other in-home energy sensors to provide users with meaningful insight into their consumption [7, 21, 65, 113, 115, 159, 169, 180, 186, 192]. Other applications process data from smartphones and wearable sensors to provide healthcare analytics [13, 66, 122]. Data from smartphone-embedded sensors is also used for many different applications that span a wide variety of user functionality [48, 72, 121]. Moreover, many applications use data from in-home sensors to perform intelligent control of actuators and appliances such as

home heating, ventilating, and air conditioning (HVAC) control [76, 79, 116, 161], and others [55, 56, 145, 169]. We refer to all such applications that make use of sensor data as *data-driven applications*.

Although applications use sensor data to provide useful functionality, they can also misuse the data to reveal private information about the users to application developers. For instance, data from motion sensors can be used to infer users' home occupancy [96]. Smart meter data can be processed to infer users' appliance usage [136] and TV viewing habits [91], socio-economic status [51], and many other types of private information [129]. Similarly, data from sensors embedded in smartphones can be used to infer a user's physical activity [101, 102], driving habits [104, 111, 147], and sleep patterns [66, 98]. Therefore, it is imperative that, while enabling such applications, *data privacy* be preserved. We define ***data privacy*** as the ability of the user to control the extent of any sharing of her sensor data with any third party [177].

Most data-driven applications today adopt a service provider-centric approach. In this approach the user is required to provide her data to the service provider in exchange for using data-driven applications. Typically, data from sensors is uploaded to the service providers' servers that host applications including their data processing, inference algorithms, and other application-specific components. Users are provided with interfaces, such as web portals, to interact with the different applications. Examples include Nest [25], Dropcam [10], and If This Then That (IFTTT) [20]. However, this approach leads to the service providers obtaining a copy of the user's sensor data, often along with the rights to share the data, and hence can share it with unauthorized third parties, such as advertisers. This threat has already come to pass in the form of unauthorized sharing of users' smart meter data [15, 24].

An alternative approach is to provide users with application executables that they host in their own compute environments such as PCs, smartphones, and tablets, which also house their sensor data. This gives the users the ability to prevent applications from relaying data to unauthorized third parties, for example, by preventing applications' access to the Internet. This approach provides better data privacy as compared to the approach discussed above because data is not transferred out of the user-controlled compute environment. However, the task of data warehousing, provisioning computational resources for applications, and enabling remote access to applications from Internet-enabled user devices, are delegated to users. These tasks put an overwhelming burden on them. Examples of such application frameworks are HomeOS [74], Beam [170], and Energy Lens [7].

In this dissertation, we propose a new software architecture that does not suffer from

the drawbacks of the existing approaches. We posit that users should be provided with personal compute environments hosted in the cloud that provide (i) persistent storage to warehouse their personal sensor data, and (ii) computational resources to run applications such as those discussed above. Applications can be downloaded from an "application store", and run within the confines of the user's ***personal virtual execution environment (VEE)***. Hosting applications on users' personal VEEs ensures that, given suitable mechanisms, no data leaves the confines of the user without her explicit permission. We describe our vision in greater detail in Section 1.3.

The remainder of this chapter is structured as follows. In Section 1.2, we identify the goals that an ideal system for personal sensor data management should achieve. In Section 1.3, we lay out the vision of our personal-VEE based architecture and outline the technical challenges in realizing this architecture. In Section 1.4, we describe the contributions of this dissertation, and conclude with a summary of the chapter in Section 1.5.

1.2 Goals

An ideal system for personal sensor data management should meet two conflicting primary goals. First, it should enable data use, that is, it should allow applications to access sensor data since applications provide users with valuable data-driven functionalities. Second, it should preserve *data privacy*, that is, it should afford users with complete control over the data that they share with each application. These two primary goals translate to the following sub-goals.

Data Consolidation

The system should provide users and applications with a single view into multiple data streams collected using different sensors. For instance, in the case of sensor data concerning users' energy consumption, the system should allow consolidation of data collected using in-home sensors and data available from utility-owned web-portals (collected by smart meters [16]). Data consolidation not only broadens the scope of possible applications by enabling cross-correlation of data, such as, correlating smart meter data with in-home sensor data, but also eliminates the need for the user to access and manage a separate data corpus for each sensor. In contrast, most commodity sensors today, for example, Fitbit [14], Dropcam [10], and smart meters, provide data access through their separate proprietary interfaces, preventing consolidation.

Data Durability

To allow applications to process historical sensor data, the system should guarantee the durability of sensor data, regardless of its time of origin. For instance, HVAC control applications [76, 79, 161] process sensor data from previous days/weeks/months.

Data Integrity

The system should ensure that sensor data is not tampered by an application or a third party such as a storage provider[1]. This requires mechanisms to detect data tampering and provide tamper evidence.

Data Privacy

The system should allow users to control who gets to access their sensor data. Therefore, the system must ensure that only the user (the data owner) has access to all data (*confidentiality*), and only the user can determine the scope and granularity of any entity's access to data (*access control*).

Application Flexibility

The system should allow a user to freely choose data-driven applications. For instance, the system should allow application developers to distribute their applications to users.

Application Performance

The system should ensure acceptable application performance, for example, ensure tolerable data retrieval times, data processing times, and application access latencies.

Scalability

The system should maintain application performance, that is, low data retrieval and processing times, and low access latencies, despite an increasing volume of sensor data, growing number of sensors, and rising number of applications.

[1] If the user employs cloud storage providers such as Windows Azure [37] or Amazon S3 [4].

We now present our proposed approach for achieving these design goals.

1.3 Our Vision

Today, a wide range of computing and storage services hosted in the cloud are available as commodity rent-based services. Examples include Virtual Machine (VM) providers (e.g., Amazon EC2 [4] and Windows Azure [37]), computing platform providers (e.g., Google AppEngine), and Blob storage providers (e.g., Amazon S3 [4] and Azure Blob Store [37]). Application developers rent these services to host their applications and users' sensor data. Users' sensor data is required to be relayed and stored with these services in order for users to use the provided applications. In doing so, the user hands over a copy of her sensor data to application developers, and in many cases the right to process and/or share the data with third parties. Thus data privacy is lost.

Our work is motivated by the following insight: if certain application hosting and storage services were available to users, it is possible to build an ecosystem of data-driven applications where data privacy is preserved. Figure 1.1 shows an overview of the application ecosystem enabled by our approach.

In this approach, each user owns and controls a virtual execution environment (VEE) hosted in the cloud, for example, a VM. Users' data is stored reliably and durably using commercial cloud storage services. Data-driven applications run within the personal VEE, and the user retains complete control over the applications, that is, controls the data reads, writes, and transfers. The personal VEE has mechanisms that a) preserve data privacy by providing users with fine-grained access control, b) prevent applications from leaking user data, and c) prevent or detect violation of data integrity by the storage provider or by an application.

Data from different sensors is consolidated in the VEE's datastore, either directly, or using a gateway such as an in-home PC. Many in-home sensors, such as *passive infrared* (PIR) sensors, currently possess limited communication capabilities or use proprietary communication protocols such as Z-Wave [39]. Thus an intermediate gateway to collect and coordinate data uploads, for example, by buffering during intermittent connectivity, is required. Nevertheless, by providing a unified consolidated view to all sensor data, our approach streamlines application development because applications can access any type of sensor data through a uniform API. We believe that, enabling correlation of different types of sensor data can lead to novel applications.

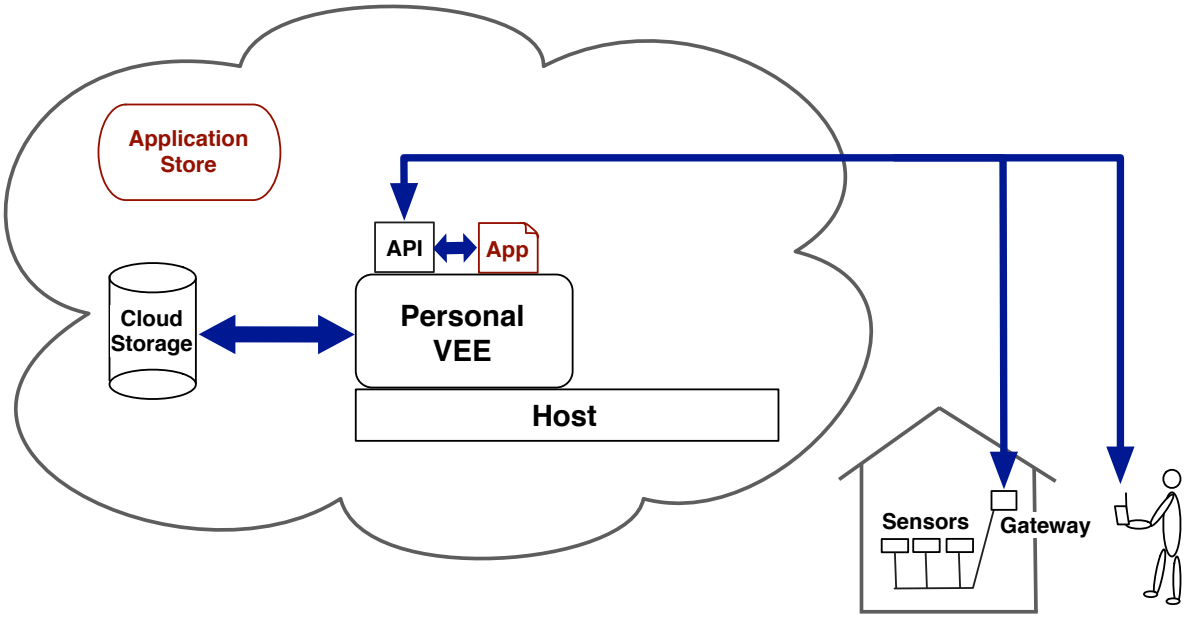


Figure 1.1: Overview of the personal VEE ecosystem.

Application developers publish their applications to an application store from which users can download and run them on their personal VEEs. This maximizes flexibility and application choice.

Users can leverage the elasticity of existing clouds to inflate or deflate their VEEs, depending on the resource requirements of the applications, thus ensuring application performance and scalability, without handling the burden of manual resource provisioning. Users may use their personal VEEs to also host private instances of web, mail, messaging, or other application servers, and exercise greater control over them than employing commercial providers of such services, such as, Gmail, Dropbox, or Google-Docs. Similarly personal VEEs can be used to offload computation of privacy-sensitive applications from resource-constrained mobile devices [90, 109, 135, 182].

Note that, in our approach the personal VEE provider is entrusted with providing a secure and private execution environment. However, unlike existing approaches, our approach does not require the user to explicitly hand over the rights to their data in exchange for using data-driven applications. Recently proposed fiduciary relationships between cloud providers and users, when legislated, will also prevent personal VEE providers from eavesdropping on user data and applications [106].

1.3.1 Technical Challenges

Many technical challenges need to addressed to realize our proposed vision. For instance, several mechanisms need to be designed and implemented to instantiate the framework. This includes mechanisms to (i) coordinate sensor data uploads with the gateway(s), (ii) expose data to third party applications via APIs, (iii) prevent applications from leaking user data, and (iv) provide fine-grained access control to users. Other mechanisms may be needed to certify and validate applications' behaviour, and to provide users with easy to understand data access controls.

Sensor data is required to be stored durably. Existing commodity cloud storage providers such as Amazon S3 [4] or Windows Azure [37], may be leveraged to do so. Although our approach requires users to trust their personal VEE provider (Section 1.3), users may not trust cloud storage providers with the confidentiality and integrity of the data. Moreover, users and applications may want to store data across multiple cloud storage providers, and some applications may also require sharing of data across homes. At the same time, time taken to retrieve stored historical sensor data should be minimized. Therefore, suitable sub-systems for storage and retrieval of data need to be built to cater to these requirements.

Our approach requires provisioning one VEE for each user[2]. Using existing virtualization solutions to provision large numbers of VEEs will require unaffordably extensive hardware resources due to relatively small VEE hosting densities of these solutions[3]. Therefore, alternative systems and methodologies to provision large numbers of personal VEEs in the cloud need to be developed.

Other challenges include simplifying the development of data-driven applications. For instance, a programming framework can be built such that it provides applications with programming abstractions and primitives for commonly used inferences such as occupancy. Applications can directly request their required inferences, while the framework can bear the onus of running the required data processing; thus greatly simplifying application development.

1.4 Contributions

This dissertation makes the following key contributions:

[2]For example, supporting each active Facebook user will require provisioning 1.4 billion VEEs [11].
[3]Citrix XenServer claims 100-200 VEEs/machine, subject to the VEEs' workload.

- We design and implement a prototype of our proposed architecture, with home energy data as the use case. We prototype the different design components and three sample applications. We conduct a qualitative comparison with existing approaches that provide applications for processing home energy data, and demonstrate that unlike existing approaches our prototype achieves the design goals described in Section 1.2.

  This work is described in detail in Chapter 3, and has been published in the proceedings of ACM e-Energy 2013 [169].

- We design a storage system (named *Bolt*) for the efficient storage and retrieval of time-series data. Bolt builds on top of untrusted cloud storage services such as Windows Azure [37] and Amazon S3 [4] to provide durability, confidentiality and integrity guarantees for sensor data. We show that, in our test scenarios, Bolt has up to 40 times lower retrieval time than OpenTSDB [27] (a popular time-series database system), and requires 3-5 times less storage space.

  This work is described in detail in Chapter 4, and has been published in the proceedings of USENIX NSDI 2014 [93].

- We propose a new methodology to multiplex large numbers of personal VEEs on a single machine at the cost of a small increase in client request latency (called *miss penalty*). We present a formal model for the problem and establish a theoretical lower bound on the miss penalty. We demonstrate the application of our methodology using LXC [22] and explore the trade-off between VEE density and miss penalty in this context.

  This work is described in detail in Chapter 5, and has been published in the proceedings of ACM VEE 2015 [167].

As we describe in the remainder of this dissertation, these contributions address different facets of our proposed personal VEE architecture (Figure 1.1), and combine to form a solution that achieves the conflicting goals of providing *data privacy* and *data use*.

1.5 Chapter Summary

Sensors have become ubiquitous. Data collected from sensors is of immense practical value. At the same time, sensor data can be processed to reveal unwarranted private information about the user. Therefore, a system architecture that enables applications of

sensor data while preserving user data privacy is required. We propose an architecture that provides users with a personal virtual execution environment (VEE) hosted in the cloud, and enables an ecosystem of third party applications of sensor data while preserving data privacy (shown in Figure 1.1). However, there are a number of challenges faced in realizing this proposed approach. Examples include building mechanisms for data collection and consolidation from sensors, preventing violation of data privacy by applications, ensuring data integrity and confidentiality when using untrusted storage services, providing efficient storage and retrieval of time-series sensor data, and feasibly provisioning personal VEEs for a large number of users. Designing systems and methodologies to address these technical challenges is the focus of this dissertation.

Chapter 2

Related Work

In this chapter we first describe an overview of existing systems that provide users with applications to process their privacy-sensitive data (in Section 2.1). We briefly describe their architecture and design, and classify them based on their architectural approach. This classification enables us to evaluate them based on the design goals described in Chapter 1. We also survey existing work that has adopted the personal VEE approach for particular applications such as online social networks, and find that this approach, when augmented with suitable mechanisms and subsystems, can achieve our desired design goals. This is the basis for the system architecture outlined in Chapter 1 and presented in greater detail in Chapter 3. In Chapter 3, we choose home energy data as an example use case, and design and implement different components of the personal VEE architecture such as an in-home gateway, a personal VEE framework, and a few sample applications.

In Section 2.2, we survey prior work on designing storage systems for user data. We find that existing systems either do not address storage and retrieval of time-series sensor data, or do not ensure data privacy and integrity when storing data using untrusted storage services. In Chapter 4 we design a storage system to meet these requirements.

In Section 2.3, we categorize different virtualization approaches and survey existing work on increasing VEE hosting density. We identify shortcomings of existing approaches which prevent them from maximizing personal VEE hosting density. In Chapter 5, we design a new methodology to alleviate these shortcomings.

2.1 Providing Data-Driven Applications

Figure 2.1 shows a taxonomy of existing systems for providing data-driven applications and Table 2.1 compares them on the basis of the design goals described in Section 1.2.

2.1.1 Service-provider Approach

In this approach, each user's data is stored on a logically centralized server, owned and controlled by a service-provider, where it is processed for analytics or for controlling actuators, or is shared with third-party applications. Most commercial data application providers today adopt this approach.

For instance, energy companies such as Waterloo North Hydro [36] and San Diego Gas and Electric [32] allow users to download their energy consumption data that is collected by utility-owned smart meters, and provide some web-based data analytics services. However, most energy companies chose to delete the data after a few months of the billing cycle.

In other systems, users have the option of uploading and optionally linking their data to other web applications. Examples include the Google Powermeter [15], Microsoft Hohm [24], and GreenButton[1] [16] applications such as Home Energy Yardstick [19], Snugg Pro Energy Audit [33], and Home Energy Saver [18]. Similarly, health-tracking services such as Fitbit [14] and Withthings [38] warehouse data from the wearable sensors on their cloud-hosted servers. Users can enable other web-hosted applications to access this data [13].

These systems do not provide any mechanisms for consolidating data from different data sources. Secondly, the user is responsible for downloading ephemeral data from the central provider and permanently storing it if they want data durability. Since data is processed on the service-providers' servers, it is vulnerable to intentional or unintentional sharing with unauthorized third parties; thus endangering data privacy. This was evident from the Google Powermeter [15] and Microsoft Hohm [24] projects, where energy companies shared users' smart meter data with third parties (Google and Microsoft respectively), without explicit user consent. Lastly, no mechanisms for ensuring data integrity are provided to the user. However, the user can chose whether to link their data to other web-applications, thus providing a limited degree of application flexibility.

[1] An initiative to standardize smart meter data formats.

Service-providers are responsible for provisioning their systems to ensure application performance and scalability.

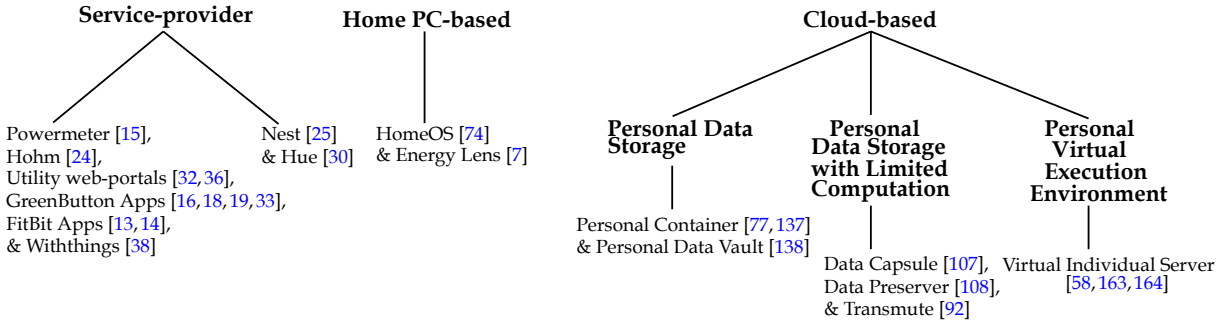


Figure 2.1: Taxonomy of existing systems for providing data-driven applications.

Sensor-oriented Services

Off-the-shelf devices such as Nest [25] upload sensor data to the service-provider's server. The server stores the data and processes it to predict home occupancy, which is used to dynamically adjusts the thermostat's set point. Similarly, the Philips Hue [30] lighting system uploads data about user preferences and usage to its server, which is processed to generate user feedback and product advertisements.

These solutions cater only to their respective sensors, and do not guarantee long-term data storage, thus providing little data consolidation and durability. Since data remains under direct control of the service-provider and not the user, data privacy is lost. Lastly, a user is limited to using the applications provided by the hardware provider; users cannot freely choose applications and thus have little flexibility.

2.1.2 Home PC-based Approach

In this approach, a user-owned computer, such as a home PC, is used to store sensor data, run data analytics, and monitoring and actuation applications. This approach is adopted by HomeOS [74], a platform for applications that use in-home sensors, and Energy Lens [7], a desktop application that analyzes users' energy consumption data.

We now provide an overview of these solutions and evaluate them based on the design goals described in Section 1.2.

HomeOS

HomeOS [74, 75] is a .NET based platform designed to provide centralized monitoring and control of sensors and actuators in a home such as switches, thermostats, and cameras. It provides application developers with homogeneous abstractions to communicate and control such devices. Figure 2.2 shows an overview of HomeOS. It is comprised of software modules called "drivers" that communicate with the sensors and actuators. A set of higher level modules called "application modules" use the driver modules to collect data from the sensors, and encapsulate algorithms to perform monitoring, actuation, or data analytics. The platform is responsible for providing (i) resource isolation between modules, (ii) sensor access control, and (iii) communication between modules.

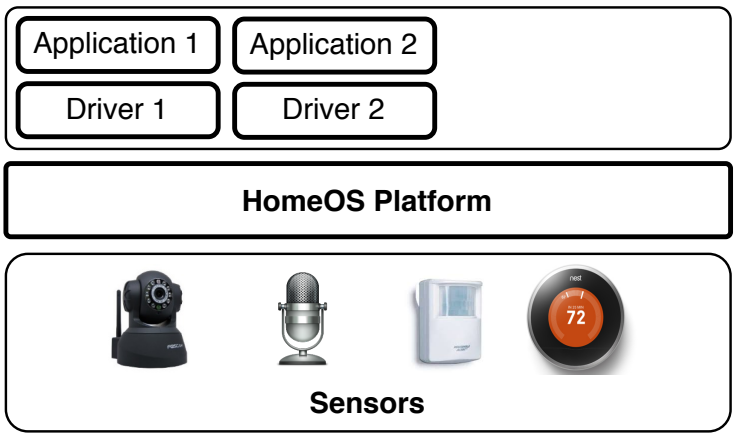


Figure 2.2: Overview of the HomeOS [74, 75] platform.

HomeOS provides data consolidation by enabling applications' to access various sensors (and other data sources such as web-portals). Since the data is processed on a user-owned PC, the user can chose which applications to install and control applications' access to sensors. Thus this solution provides data privacy and integrity, and application flexibility and extensibility. However the user is required to maintain the durability of the data, for example, using RAID arrays and off-site backups. Applications' performance in HomeOS is limited by the computational resources of the host computer, and

the user must provide additional computational and storage resources as application demands and data volumes increase.

Energy Lens

Energy Lens [7] is a desktop application for homeowners and energy auditors that processes energy data and generates feedback in the form of interactive visualizations. It helps users in keeping track of when, where, and how much energy is wasted, and progress made in reducing energy consumption. Similar to HomeOS, it allows users to consolidate and process energy data from different sources (in-home sensors and smart meters), preserve data integrity, privacy, and allows extensibility and flexibility. However, this approach provides no mechanisms for maintaining data durability, application performance, or scalability.

2.1.3 Personal Data Storage

In this approach, each user is provided with a cloud-hosted storage system instance controlled by the user herself. Users can use their individual storage environment to consolidate and archive their data, and can chose to share it with third party applications (hosted elsewhere in the cloud). We now describe existing work that has proposed this approach.

Personal Container

The Personal Container [137] is a user's personal data storage system instance in the cloud, a logical single intermediating entity, where data is consolidated from disparate sources such as various user-owned sensors, and re-exposed to data-consumers such as applications, subject to the user's consent. Figure 2.3 outlines the Personal Container ecosystem of users, data sources, and applications.

Elsmore et al. [77] demonstrate the use of Personal Containers for collecting a user's smartphone GPS data. They system is deployed in an organization that wishes to measure its employees' travel-to-work carbon footprint, for planning future infrastructure. The Locker project [35] is an open-source platform that allows users to store information and create applications that can run inside *lockers*. Each Personal Container is initialized as a locker instance running on a Linux Container (LXC) [22], where each user's GPS

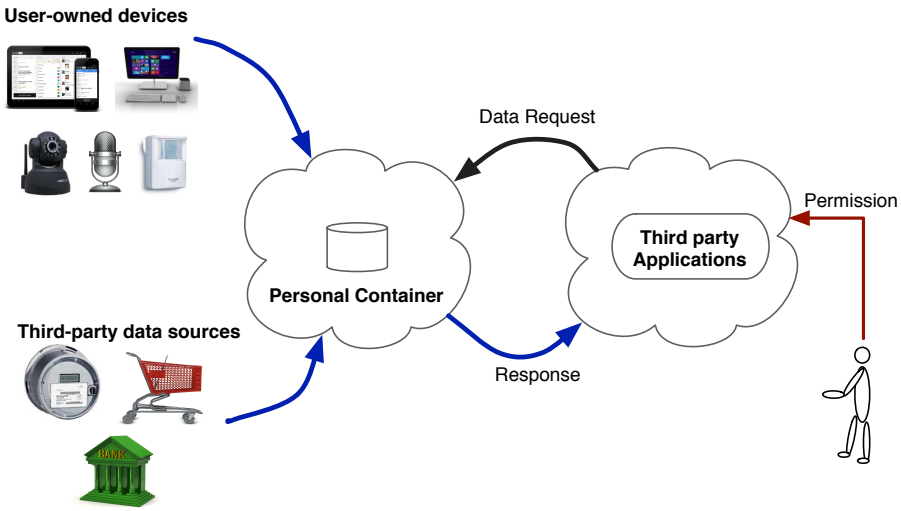


Figure 2.3: Overview of the Personal Container approach [137].

data is stored. The organization is then able to access some parts of users' data with their explicit permission, and offers data-driven applications. Example applications include one that calculates how much money a user spends using a car, train, or bus during their commute and computes the payback time of using the bike-to-work scheme to buy a bicycle.

Personal Containers allow users to consolidate data from different sources and durably store it. Users have a free choice over applications, and application providers are responsible for maintaining application performance and scalability. Users are required to trust the Personal Container with maintaining the integrity of the data. However, data is transferred out of the Personal Container to application servers, albeit with explicit user permission, thus giving up data privacy. Moreover, existing work [77, 137] does not address efficient storage and retrieval of sensor data and provisioning of Personal Containers for large numbers of users.

Personal Data Vault

Similar to Personal Containers [77, 137], Mun et al. [138] propose *Personal Data Vault*, a system to allow users to retain ownership of their data. The proposed solution is to use secure user-owned data containers to enable user-controlled sharing of data with applications. This approach uses three mechanisms to facilitate this selective sharing

of data with applications. First, *granular access control lists*, allow users to define data sharing policies that include the type and granularity of data a given application can access. Second, *trace-audit* logs record transactions and transformations of users' data performed by applications. Lastly, a *rule recommender* pre-calculates constraint values for a set of pre-defined data sharing policies, and facilitates policy re-configuration. The system is instrumented to collect data from users' smartphones for two health monitoring applications: *Ambulation* (which logs a user's physical activities) and *Walking-survey* (which records a user's sleep cycle information).

Similar to the Personal Container system, this solution achieves all of our design goals except providing data privacy, since data is shared with the service-provider, albeit in a controlled fashion. Moreover, this work does not address provisioning of Personal Data Vaults for large numbers of users.

2.1.4 Personal Data Storage With Limited Computation

This approach enhances the personal data storage approach by supporting some transformations of data before sharing with third parties; thus requiring a limited amount of computation to perform such transformations.

Transmute

Griffis et al. [92] propose *Transmute*, a novel scripting language for writing data parsing scripts that can be run inside a Personal Data Vault [138]. Transmute scripts or *filters* receive raw sensor data streams as input, and output privacy-preserving data streams, which can be shared with untrusted applications. This reduces the amount of private user information revealed to the application. For instance, a filter can convert smartphone GPS data to a more coarse grained data stream such as a ZIP code stream. The authors also provide tools that can be used to analyze the filters, and provide guarantees about the transformations they perform on a given input stream. Users are also provided with access control lists which can be used to grant stream read/write privileges to filters and applications. Filters are sandboxed to prevent malicious filters from leaking data.

This solution retains the data consolidation, integrity, and durability benefits of the Personal Data Vault approach. In addition, it addresses data privacy using its filter-based data transformations. However, it precludes applications from using raw data

streams. This presents serious limitations for application developers because they need to design and implement appropriate methods for using transformed data streams, thus limitting application extensibility and flexibility. Moreover, filters need to be developed and maintained for every sensor type, for example by a trusted filter provider. Filters need to be continually updated (by a trusted entity) as new algorithms that reveal unsanctioned private information from transformed data evolve.

Data Capsule

Kannan et al. [107] propose *Data Capsule*, an encapsulation of a specific kind of user data such as credit card numbers or web-browsing patterns, along with code that implements a well-defined and open interface for accessing the data, for example, an interface that debits money from a given credit card. Applications hosted on service-providers' servers can use these interfaces to perform operations using the data, without transferring data out of the user-owned capsule. The framework includes a policy layer that allows a user to exercise fine-grained and flexible control over how the interface to her data is used. It also allows users to transform data before it is shared to limit loss of data privacy. The solution demonstrates a shopping application in which the capsule provides an interface to charge a credit card without revealing the credit card number to the application. Similarly, in a second application the capsule stores a user's web-browsing data, and applications query the capsule interfaces to generate user-targeted advertisements, and the application does not access the raw data.

Similar to the previous solution, this solution provides data consolidation, integrity, and durability, and preserves data privacy by regulating applications' access to data using capsule interfaces. However, it requires development and maintenance of capsule interfaces for different data types, and for different data processing operations that applications may require. This presents a serious limitation to application flexibility and extensibility. Lastly, existing work [107] has not addressed the design of capsule interfaces for sensor data and its processing.

Data Preserver

Extending the Data Capsule framework [107], Kannan et al. [108] propose *Data Preserver*, an entity that encapsulates the user's data with code and policies chosen by the user. The framework proposes a combination of different mechanisms such as administrative isolation, software-based isolation using virtual machines, and hardware based trusted

	Service-provider	Sensor-oriented services	Home PC-based	Personal data storage	Personal data storage with limited compute	Personal VEE
	Powermeter [15], Hohm [24], Utility web-portals [32, 36], GreenButton apps [16, 18, 19, 33], FitBit Apps [13, 14], Withthings [38]	Nest [25], Hue [30]	HomeOS [74, 75], Energy Lens [7]	Personal Container [77, 137], Personal Data Vault [138]	Data Capsule [107], Data Preserver [108], Transmute [92]	VIS [58, 163, 164], πbox [124]
Consolidation			✓	✓	✓	✓
Durability				✓	✓	✓
Integrity			✓	✓	✓	✓
Data privacy			✓		✓	✓
Flexibility	✓		✓	✓		✓
Scalability	✓	✓		✓	✓	✓
Application Performance	✓	✓		✓	✓	✓

Table 2.1: Comparison of existing solutions for providing data-driven applications based on the design goals.

platform modules to guarantee that applications interact with the preserver only via the user-configured interfaces. The solution is similar in nature to the Data Capsule framework, achieves the same design goals, and suffers from similar drawbacks.

2.1.5 Personal VEE

We now describe an approach where each user owns and controls a personal execution environment hosted in the cloud, such as a virtual machine. As shown in Table 2.1, this approach meets all our design goals. We now describe existing work that has adopted this approach, and its shortcomings.

Virtual Individual Server

Cáceres et al. [58, 163, 164] identify various problems in using a centralized service-provider approach for applications that process data collected from sensors on users' smartphones such as GPS data. First, concentrating data from multiple users under one administrative domain leads to the possibility of large-scale data privacy breaches. Second, users are required to grant service providers with the rights to display and freely distribute their personal data. The authors posit that a user's interests are best served if her data is uploaded to a *Virtual Individual Server (VIS)* for two reasons: (i) VISs are resistant to large-scale privacy breaches because each VIS runs in its own administrative domain, and (ii) VISs provide users with control over software and policies that direct

all data sharing. They use this approach to design a location-based online social network application that uses virtual machines hosted on Amazon EC2 [3] as VISs. Further, they show that the application's latencies are comparable to that when using a centralized service-provider approach.

Unlike previous approaches, VISs do not require any additional interfaces or filters to be developed and maintained, promoting application extensibility. Users can completely control applications' data access. By leveraging an elastic utility computing infrastructure, VISs ensure that applications have sufficient compute resources, thus providing application performance and scalability. This approach is the foundation that we build upon in our work.

Note that existing work [58, 163, 164] uses the VIS approach for a single application, and does not address extending this approach to other possible data-driven applications. Moreover, it instantiates VISs as virtual machines, uses virtual machines' local disk to store data, and does not address durable storage of data and ensuring data integrity. Our work in Chapters 3 and 4 addresses these shortcomings.

πbox

πBox [124] extends the VIS approach by proposing a runtime for smartphone applications that spans a cloud-hosted execution environment and user devices. However, this work does not address data consolidation from other user-owned sensors, durable storage of data, and ensuring data integrity. Nevertheless, it provides mathematical bounds on the amount of information an application that reports usage statistics to developers' servers can reveal. This work is complementary to our vision (outlined in Section 1.3), because it enables data-driven applications running on users' personal VEEs to report usage statistics to developers.

2.2 Providing Data Storage and Integrity

We first discuss systems that leverage untrusted remote storage services that provide data storage with data integrity guarantees (in Section 2.2.1). In Section 2.2.2, we describe existing systems aimed at personal data storage across user devices. Lastly, in Section 2.2.3 we discuss other storage systems such as OpenTSDB [27] (which we use in Chapter 4 as a point of comparison).

2.2.1 Leveraging Untrusted Remote Storage

Li et al. [126] build SUNDR, a network file system that uses untrusted remote storage services to store files and provides integrity and consistency guarantees for stored files. It employs cryptographic mechanisms such as digital signatures and fork consistency to protect file system contents and enables clients to detect any unauthorized modifications. The authors show that a malicious user with complete administrative control of a SUNDR server cannot cause the clients to read altered contents of stored files. Using example workloads such as the Concurrent Versions System (CVS), they demonstrate that the latency overhead of SUNDR is minimal as compared to NFS [155].

Extending the approach in SUNDR [126], SPORC [80] provides a "generic collaboration service" using a cloud-hosted server running on a VM. The service is used to instantiate different applications such as a key-value store and a collaborative text editor. The design of the service ensures that the cloud-hosted server only receives encrypted data. It uses "fork* consistency" and "operational transformation" mechanisms to enable clients to detect unauthorized operations such as additions, modifications, deletions, or re-orderings. The authors use the example workloads to demonstrate that client latency overheads are negligible.

Venus [166] provides key-value storage using commodity cloud storage services such as Amazon S3 [4]. It does not require users to rent cloud based virtual machines for server hosting (as in SPORC [80]). More importantly, it does not require users to trust storage service providers, provides data integrity and consistency guarantees, and detects any data tampering by a service provider. Similar to Venus, Depot [132] provides a key-value store which although using untrusted storage services, provides consistency, staleness, durability, and recovery properties.

Farsite [45] proposes using a collection of insecure and unreliable desktop computers to instantiate a reliable virtual file server with limited data integrity guarantees. Chefs [85] demonstrates content distribution by replication of an entire file system on untrusted storage servers.

Although this body of work demonstrates the use of different techniques to leverage untrusted storage services (including commodity cloud storage services), it does not address efficient storage and low-latency retrieval of time-series data.

2.2.2 Storing Data Across Devices

Salmon et al. [154] propose Perspective, a semantic file system to help users in managing their data spread across their personal devices, such as laptops and smartphones. It provides a uniform abstraction called a *view*, which is a semantic description of a set of files. It is specified as a query on file attributes and the IDs of devices on which the files are stored. Multiple devices interface with each other in a peer-to-peer (P2P) fashion, to provide the unified view abstraction.

Similar to Perspective [154], HomeViews [87] provides users with a view abstraction, and also allows secure sharing of views across user devices. It works in a P2P fashion and does not require users to manage a centralized account or data protection mechanisms. Both Perspective [154] and HomeViews [87] target user data such as photos, music, and documents, stored across user devices.

Goh et al. [89] propose SiRiUS, a system that supplements local device file system storage with untrusted remote storage. It forms an overlay over local file systems and other network and P2P file systems such as NFS [155] and OceanStore. It provides some data integrity and confidentiality guarantees, however, it does not guarantee the freshness of data a reader receives. Similarly, VStore++ [109, 110] provides a "virtual store" abstraction, which is dynamically mapped to suitable user devices or cloud storage.

This body of work addresses many challenges associated with managing data produced and consumed by applications across user devices. However, it does not specifically address storage and retrieval of time-series data. Moreover, it does not focus on data confidentiality and integrity guarantees for such data.

2.2.3 Other Systems

sMAP [73] explores the design of a RESTful service that mediates data transfers between sensors (the data sources), and data consumers such as applications. It provides interpretability of data streams, manages the consolidation and propagation of data from sensors to a given consumer, and focuses on supporting a wide variety of sensors. However it does not focus on efficient storage and low-latency retrieval of sensor data.

Stream processing systems also present alternative solutions for housing sensor data. They enable data to be pushed through a data-processing subsystem, offer *straight-through* processing, that is, no modification to incoming data messages, which are then stored. Examples of such systems include Aurora [61] and Borealis [43]. However, these system assume that entities such as the data writer, data reader, and storage servers, are

in a single administrative domain. Scenarios where users want to leverage commodity storage services such as Amazon S3 [4] or Windows Azure [37] are therefore not supported. Specialized time-series databases such as OpenTSDB [27] (described next) also suffer from this drawback.

OpenTSDB [27] is a popular time-series data storage system. It builds on top of HBase [6] (a column oriented database management system), and exposes a data stream abstraction for reading and writing sensor data. Applications using OpenTSDB can store and retrieve data streams by using its HTTP-based web APIs. However, OpenTSDB does not provide any data integrity guarantees. This leaves data vulnerable to modifications by malicious third parties such as a service provider hosting OpenTSDB instances.

Ming et al. [127] propose a framework for storage and retrieval of patient health records (PHR). The framework divides users into subsets, and provides each subset with a separate set of cryptographic keys for encrypting data before uploading to a cloud server. Using attribute based encryption techniques, the authors demonstrate reduction in key distribution complexity with support for revocation of user access rights. However, this work relies on cooperation from the cloud hosted server (while assuming it to be non-malicious), and does not focus on time-series data.

Popa et al. [144] design a cloud storage system which provides users with data integrity and freshness guarantees. However, they do not address storage of time-series data, and do not focus on leveraging existing commodity cloud storage services.

2.3 Hosting Large Numbers of Personal VEEs

We first provide an overview of different approaches used to create virtual execution environments (VEEs) in Section 2.3.1. In Section 2.3.2, we discuss existing work that has focussed on increasing VEE hosting density, and examine its applicability for creating large numbers of personal VEEs. Lastly in Section 2.3.3, we identify *just-in-time* provisioning of VEEs as a potential approach to increase hosting density, and discuss existing work and open problems pertaining to this approach.

2.3.1 Virtualization Approaches

Virtualization refers to the combination of indirection and multiplexing to present applications with the illusion of running on a dedicated physical machine despite sharing the

physical machine's resources with other applications. Many virtualization approaches have been studied since the mid-1960s and can be broadly classified into three categories: Full, Para-, and OS-level virtualization. Table 2.2 shows example virtualization solutions for these categories.

Full Virtualization

In this approach, a single physical machine simultaneously hosts multiple unmodified operating systems by trapping and redirecting requests to the underlying hardware resources, including the CPU, interrupts, and the memory subsystem. Well known examples include IBM's zServer, and VMWare Server. This approach places the most demands on the physical machine, and therefore has the least potential VEE hosting density, but provides the best fault and performance isolation.

Para-Virtualization

This approach presents the same interface to applications as full virtualization, but makes small modifications to the hosted operating systems to reduce their resource demands. This technique was first introduced in the Xen [50] virtualization solution. This approach provides the same fault and performance isolation as full virtualization and achieves a greater potential density, but at the cost of requiring OS modifications.

OS-level Virtualization

This approach virtualizes the API between an application and the OS, and makes a single instance of an OS appear like multiple OS instances. The key idea is to tag a set of processes with an identifier and to mutually hide processes with differing tags from each other at every level of the operating system. By running a single instance of the OS, this approach greatly reduces resource demands and increases hosting density. However, this density improvement is at the cost of requiring OS modifications, and reduced fault isolation, in that a kernel failure may simultaneously crash all virtual OS instances. A widely used solution that has adopted this approach is Linux Containers (LXC) [22].

The VEE hosting density of a given virtualization solution directly depends on its resource overhead. Therefore, to maximize VEE hosting density, it is imperative to understand and compare the resource overheads of the different virtualization approaches.

Type	Solutions
Full Virtualization	VirtualBox, VMWare Player, VMWare ESX, LPAR, QEMU, MoL
Para-Virtualization	Xen, KVM, Lguest, rhype, UML, L4Linux, z/VM, PHYP, lv1, BEAT
OS-level Virtualization	OpenVZ, Linux-VServer, LXC

Table 2.2: Examples of existing virtualization solutions.

Resource Overhead Comparison of Virtualization Approaches

Para-virtualized approaches have been shown to incur lower resource overhead than fully virtualized approaches [50], thus they can be hosted with greater density. OS-level virtualization incurs even lower overhead than para-virtualization and therefore allows even higher VEE density. Soltesz et al. [171, 172] use benchmarks such as lmbench, Operf, DBench, Postmark, OSDB, IPerf, and SPECWeb99, to demonstrate that the OS-level virtualized VServer approach incurs lower resource overhead than the Xen paravirtualized approach. Other researchers comparing OS-level virtualization solutions with para-virtualized approaches report similar findings [63, 64, 143, 175]. Therefore, for a given workload, OS-level virtualization solutions can potentially yield higher VEE density.

Moreover, the resource overheads of competing OS-level virtualization solutions is similar. For example, VServer and OpenVZ show similar resource overheads, and hence similar VEE hosting densities [60].

2.3.2 High Density Hosting

An ideal virtualization solution for provisioning personal VEEs should meet the following goals.

- It should allow users to freely run multiple applications on their personal VEEs.
- It should minimize any impact on the performance of applications hosted on personal VEEs, for example, minimize any increase in latency of clients accessing application servers hosted on a VEE.
- It should maximize VEE density, that is, the number of personal VEEs co-hosted on a single machine.

In light of these goals, we now discuss existing work on increasing VEE hosting density.

The Denali isolation kernel [181] was one of the first attempts to create a high density VEE hosting solution. It achieved an astonishing 10,000 VEEs per machine. However, each VEE may only host a single application run on behalf of a single user. Moreover, it requires both the guest OS and applications to be modified, thus requiring modifications with the release of each application update. Therefore, this approach is not suitable for mass hosting of personal VEEs.

The Honeyfarm approach [178] modifies Xen so that a VEE is dynamically spawned whenever a client request (for a server hosted on the VEE) arrives, and is terminated when the request is fulfilled. This work proposes two key improvements to Xen: (i) *flash cloning*, which is a mechanism to quickly instantiate new VMs by copying and modifying a reference VM image, and (ii) *delta virtualization*, which is a copy-on-write scheme for VM main memory. This approach demonstrates VEE density of 10,000 VEEs per machine. However, it does not allow data persistence across client invocations: all VEE state is lost on the termination of the client request, making it unsuitable for hosting personal applications, such as data analytics. Consequently, this approach is unsuitable for hosting personal VEEs. Nevertheless, as we see next in Section 2.3.3, the idea of just-in-time provisioning of VEEs (used in this approach) greatly helps in improving VEE hosting density.

2.3.3 Just-in-time Provisioning of VEEs

Traditionally, VEEs provisioned on a machine are thought of as always being in a booted state, and thus utilizing the host's CPU, memory, and disks [71, 103, 123, 190]. However, many VM workloads exhibit frequent, often long, and uncorrelated idle periods. Examples include personal VEE workloads, some web-hosting [118] and cyber-foraging workloads [158]. When multiple VEEs with such workloads are co-hosted on a machine, decreasing the resource footprint of idle VEEs allows for a much denser VM packing, thus reducing hosting costs for both VEE renters and VEE providers.

It is possible to reclaim resources from an idle VEE by transitioning it to an inactive state(s), where its resource footprint is reduced, and activating it at the end of its idle period, for example, on the arrival of a client request for a server hosted on the VEE. The increase in VEE density comes at the cost of an increase in client request latency, called the *miss penalty*.

Inactive States

Table 2.3 shows inactive states proposed or supported in a few virtualization solutions. Recent work has introduced inactive states for VEEs to reduce their resource footprint, albeit for different reasons, such as, for reducing VEE activation time. Wang et al. [179] propose stateful in-memory *substrates* which are less resource-intensive than a running VEE, and have small activation times. Knauth et al. [117] propose a fast-resume state which leverages lazy disk reads to lower VEE activation time. Likewise, Twinkle [191] demonstrates the use of different optimizations to lower VEE resume (from suspended) times, such as, working set estimation, demand prediction, and free page avoidance .

Virtualization solution	**Inactive states**
LXC [22]	Frozen(stock) [133], Shutdown (stock)
Xen	Suspended, Shutdown (stock), Substrates [179]
VMWare ESXi	Suspended, Shutdown (stock), Fast-resume [187,188]
KVM	Suspended, Shutdown (stock)

Table 2.3: Proposed and natively supported (denoted "stock") inactive states for a few virtualization solutions.

Increasing VEE Density

Existing work has explored the use of a single inactive state for hosting idle VEEs, and increasing VEE hosting density. DreamServer [118] demonstrates the use of a suspended state [117,187,188] for just-in-time provisioning of VEEs for web-hosting workloads exhibiting idle periods. The authors demonstrate that hosting density can be increased by up to 50% by transitioning idle VEEs to the suspended state, and activating them on the arrival of client requests. Similarly, Ha et al. [94] explore just-in-time provisioning for VEEs that run computations offloaded from resource-constrained mobile devices. This body of work suffers from two main drawbacks.

First, existing work uses an on-demand policy for managing idle VEEs. That is, a VEE is transitioned to the booted state only when a client request for it is received, thus incurring the maximum miss penalty. However, miss penalties can potentially be minimized by leveraging better policies for managing idle VEEs, for example, by proactively

activating a VEE in anticipation of client requests. Unfortunately, existing work has overlooked the design of such policies, and their impact on miss penalties.

Second, existing work leverages only a single inactive state. This limits VM density to the maximum number of inactive VEEs that can co-exist in that particular state. However, since inactive states differ in their resource requirements, VEE density can be further improved by multiplexing idle VEEs across multiple inactive states. Unfortunately, existing work has overlooked this potential avenue for increasing VEE density, which presents significant density improvements for workloads with high idle periods such as personal VEEs. We address these two drawbacks of existing work in Chapter 5.

In using such a just-in-time provisioning approach, two key mechanisms are required:

- Determining when a VEE has become idle, so that it may be transitioned to an inactive state.
- Detecting when a VEE needs to become active, so that it can be transitioned into the booted state, for example, on the arrival of a client request.

We now describe prior work in these two areas.

Determining VEE Idleness and Activeness

The amount of time a VEE is idle depends entirely on its workload. For instance, if a VEE hosts application servers, it may be classified as idle once it has no outstanding client requests. Existing work has shown how such rules to determine VEE idle time can be created. For instance, in case of client-server workloads such rules can use the number of connected TCP clients [168], or the VEE's CPU and memory utilization [184]. Example mechanisms to implement such rules include *VEE introspection* [100, 184] or a reverse-proxy server running on the host machine [118].

Some of these mechanisms can also be used to trap client requests and activate respective VEEs on request arrival. Existing work has demonstrated that this can be accomplished using a reverse-proxy server [118]. Other possible mechanisms (overlooked in existing work) include deploying a kernel module on the host machine which uses the destination IP address in a request to identify and activate the target VEE.

An alternate solution is to use a customized DNS server. Most client requests are likely to use TCP along with using the target VEE's domain name to address it. Therefore

a DNS server running on the host machine can be used to trap client requests and activate the target VEE. VEE introspection can then be used to monitor a VEE's active client connections and detect when a VEE becomes idle, that is, it has no established client connections. Moreover, this mechanism can also be used to multiplex a small pool of IP address among a comparatively larger number of VEEs [168].

2.4 Chapter Summary

Upon comparing different approaches for enabling data-driven applications, we find that existing approaches do not meet our desired design goals. However, a careful analysis of prior work suggests that an approach that provides each users with her own personal VEE in the cloud can potentially meet all the desired design goals. To realize this approach for a large number of users, it needs to be supplemented with several subsystems. This includes an in-home gateway for collecting sensor data, building a framework for hosting data-driven applications on personal VEEs, and providing data access control. Our work in Chapter 3 addresses the design and implementation of such mechanisms.

A suitable data storage system for efficient storage and retrieval of time-series data also needs to be designed. Commodity storage services may be leveraged to do so. However, they may not be trusted with maintaining the confidentiality and integrity of the data. Existing approaches to build storage systems for user data either do not focus on time-series data, or do not provide the required data confidentiality and integrity guarantees. Nevertheless, many techniques used for interfacing with untrusted storage services can potentially be leveraged to build a storage system for time-series sensor data. We address this problem in Chapter 4.

Similarly, provisioning personal VEEs for a large number of users requires suitable virtualization solutions and methodologies to be developed. Existing work has proposed a just-in-time VEE provisioning approach for increasing VEE density. In this approach the resource footprint of idle VEEs is reduced by transitioning them to an inactive state. VEEs are activated on-demand, for example, upon the arrival of a client request. The VEE density improvement thus is at the cost of an increase in client latency. However, existing work has overlooked venues for further improving VEE density by multiplexing VEEs across multiple inactive states. Also, the increase in client latency can potentially be lowered by using different policies for managing idle VEEs across inactive states. Our work in Chapter 5 addresses these drawbacks.

Chapter 3

Leveraging Personal VEEs for Energy Data Analytics

3.1 Introduction

As discussed in Chapter 2, existing approaches for enabling data-driven applications do not meet the design goals described in Chapter 1. We also observe that the personal VEE approach, when supplemented with appropriate mechanisms, can potentially meet the design goals. In this chapter, we design and implement the required mechanisms to instantiate the personal VEE architecture, determine if our proposed system meets the design goals, and thus examine the feasibility of our approach for enabling privacy-preserving data-driven applications.

The variety of areas in which data-driven applications are employed today is extremely wide, ranging from healthcare, safety, convenience, to energy efficiency. To avoid being overwhelmed by domain-specific details, we choose *home energy* as an example use case, and build a prototype implementation of the personal VEE architecture that hosts users' home energy consumption data. We choose home energy data as our example use case because of the following observations.

- Smart meters are being increasingly deployed in homes across the world [41, 42], thus providing users with whole-house energy consumption data.
- The availability and adoption of commodity home energy sensors and actuators have grown tremendously in recent years [3], thus enabling easy instrumentation of home appliances for monitoring and control.

- Home energy data can be used to provide users with meaningful and actionable insight into their consumption habits [7, 21, 65, 113, 115, 159, 169, 180, 186, 192].
- Suitable processing of home energy data can reveal private information about the users, such as their appliance usage habits [136], TV channel being viewed [91], socio-economic status [51], and other private details [129].

We believe that these recent trends speak to the applicability, value, and potential impact of our work.

We note that a large body of recent work has focused on designing theoretical schemes for analyzing smart meter data in a privacy-preserving fashion. Example approaches include obfuscation [44, 114], aggregation [152, 153, 165], homomorphic encryption [86, 125], and other schemes [136, 148]. Other work has quantified the utility and privacy of smart meter data, and has modelled the trade-off between them [146]. This body of work further motivates us to focus on the domain of home energy data.

In this chapter we present the design and implementation of the different components of the personal VEE architecture outlined in Chapter 1 (illustrated in Figure 1.1). These include a) an in-home gateway for relaying sensor data and control commands, b) a personal VEE framework, called *VHome*, which warehouses home energy data, exposes it to third-party applications via APIs, and provides data access control, and c) sample home energy applications that run within an instance of VHome. Our VHome implementation also enables applications to securely and privately control home appliances, when permitted by the user.

Our key contributions in this chapter are:

- The architecture of a system that allows users to own and control access to their home energy consumption data, and freely use data-driven applications of their choice.
- A prototype implementation of the proposed architecture on modern commodity cloud computing platforms, along with four sample home energy applications.
- A qualitative evaluation of the prototype implementation with respect to data privacy and data use.

This work has been published in the proceedings of ACM e-Energy 2013 [169].

The remainder of the chapter is structured as follows. We describe the system architecture, prototype implementation details, and a qualitative evaluation in Sections 3.2, 3.3,

and 3.4 respectively. We present a discussion of this work in Section 3.5 and conclude with a summary of the chapter in Section 3.6.

3.2 System Architecture

Figure 3.1 shows an overview of our system architecture. It has four main components (a) in-home *gateway* (labelled Gateway), (b) the *VHome* framework running on a personal VEE in the cloud, (c) *native applications* running within a VHome instance, and *cloud-based applications (CBAs)* hosted in the cloud by respective providers, and (d) *User interfaces* (labelled Remote UIs) that enable accessing VHome services from a connected device.

We now describe each component in greater detail.

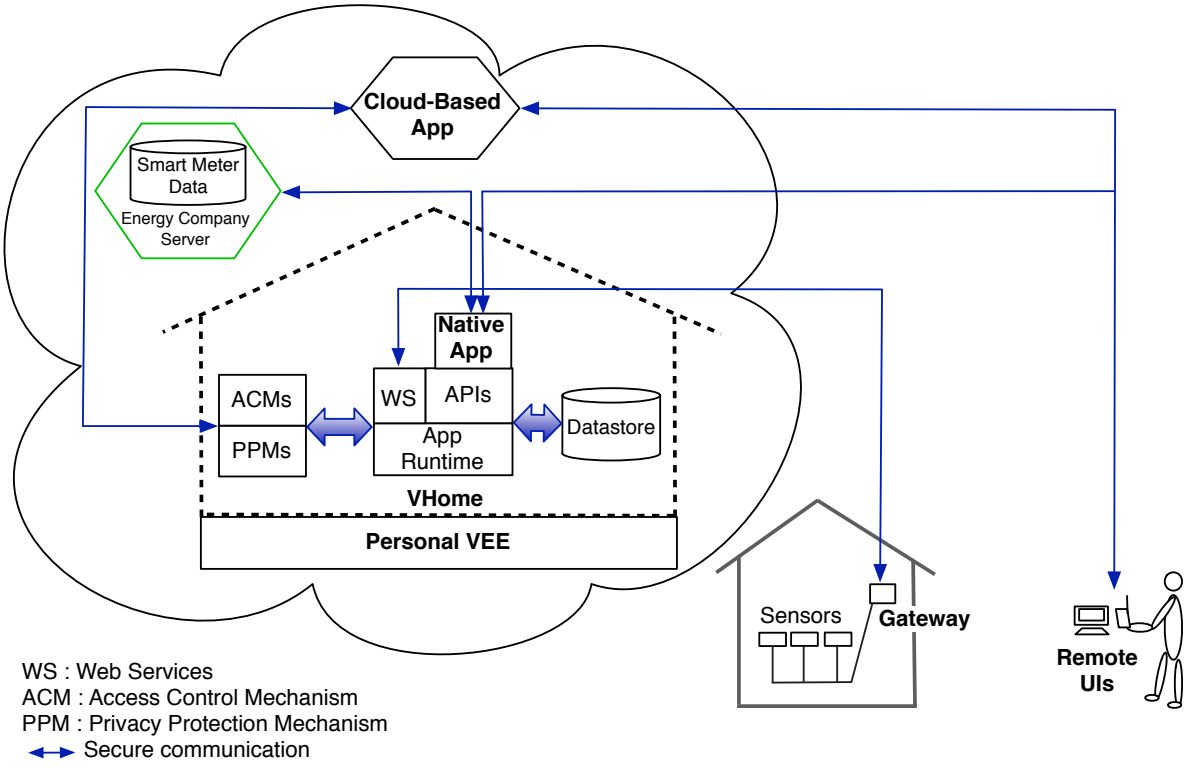


Figure 3.1: Overview of the proposed system architecture.

3.2.1 Gateway

The gateway is an in-home and user-controlled service that serves two main purposes. First, it collects data from in-home sensors and relays it over a secure connection to the cloud-based VHome instance. Second, it provides an interface that applications (and hence users) can use to control devices in the home.

The gateway interacts with appliances, either directly (in case of smart appliances), or using add-on sensors such as network-enabled power strips. This communication typically uses one or more types of channels such as USB, Zigbee, Ethernet, WiFi, or Z-Wave [39]. Usage data is uploaded from the gateway to the VHome over a secure communication channel, where it is warehoused to be accessed and processed by applications. The gateway also authenticates remote users and accepts control commands. These control commands either configure the gateway, request data uploads, or request actions to be taken by appliances and devices such as turning an appliance off. A gateway may be a dedicated, networked hardware device, or integrated into other in-home hardware such as a cable modem or set-top box, or may be software deployed on a household computer.

3.2.2 VHome

VHome is a software framework that is hosted on a personal VEE and is comprised of a set of services, described next. A VHome instance (a) receives data relayed from the gateway and stores it (either using a local datastore, or using commodity cloud storage services), (b) implements a set of APIs to access the data, (c) hosts a runtime for executing third-party applications that process sensor data, (d) implements a set of trusted web services for interaction with the gateway, and user devices, and (e) implements access control and data transformation mechanisms (explained below). A VHome instance is controlled and configured by the user who is the owner of the personal VEE.[1] A personal VEE could be a virtual machine [50] or a virtual container [171,172], provisioned by cloud providers. In Figure 3.1, the VHome is shown as a dashed outline of a home.

We classify data-driven applications into two categories: Native applications, and Cloud-based applications (CBA). *Native* applications run on the VHome-provided runtime, and are certified to be "safe" using an approach described in more detail in Section 3.2.3. In contrast, *cloud-based applications* transfer sensor data out of the VHome. Since

[1] We discuss the participation incentives for the user and the personal VEE providers in Section 3.5.

this may violate user data privacy, data accesses by a CBA are mediated by *privacy protection mechanisms* (PPMs) that pre-process data before it is transferred out of the personal VEE.

PPMs can implement privacy models such as obfuscation [44, 114], noise addition, aggregation [152, 153, 165], and homomorphic encryption [86, 125]. An example PPM is one that adds random noise values to smart meter readings, with the amplitude of the noise decreasing with reading granularity, so that monthly readings may have little or no added noise, but per-second readings have relatively larger amounts of added noise. In addition, access control mechanisms (ACMs) allow users to restrict and revoke CBA access to the APIs by scope and duration. For example, an ACM may allow a CBA to access only hourly smart meter readings for a specified day of the year. Moreover, this access may expire after one day.

In addition to native applications, a VHome also hosts special-purpose trusted *Web Services (WS)*. Being a trusted component, these services have free access to the APIs and hence to the sensor data. They periodically accept data batches uploaded from the gateway and store it in the VHome datastore. They also fetch real-time data from the gateway when requested by the user. This allows data to be uploaded in a single batch (e.g., once a day), while providing real-time data access when required. Moreover, they provide a control interface to the user for various administrative tasks, such as downloading and running native VHome applications, configuring ACMs and PPMs, requesting VHome software updates, migrating or deleting data, and configuring gateway actions.

3.2.3 Applications

We now discuss native and cloud-based applications in greater detail. Note that, the main difference between native and cloud-based applications is that native applications execute on a tightly-controlled runtime. Moreover, if applications are written in a managed language such as Java or C#, their bytecode is available for analysis. This allows the system to limit the privacy leakage that is possible due to these applications. In contrast, cloud-based applications cannot be tightly controlled. Therefore, the only way to preserve privacy when transferring data to these applications is to modify the data itself, which is accomplished using the PPMs.

We envision that both classes of applications would be developed by third-party developers, much like those who participate in the Apple App Store or Google Play Store. Developers would use the VHome APIs (described in the next section), to access sensor data. Users can either download native applications to the VHome which run on the

VHome runtime, or can use ACMs to give cloud-based applications access to their data (after processing by PPMs). Applications can provide user interfaces (UIs) to enable their invocation from PCs, smart phones, or other connected devices.

Native Applications (NAs)

The leakage of sensor data from native applications can be restricted using one of the following approaches. In the first approach, a native application's bytecode is scanned to ensure that the application is incapable of network communication. Therefore an application cannot leak data out of the personal VEE, thus guaranteeing data privacy. In our prototype implementation, native applications must be written in Java and are not allowed to invoke native APIs. Our thinking is that, an application can be certified as safe if its bytecode does not use the Java.Net API (and does not use any native APIs). This can be easily checked either at application installation time or when it is submitted for publication to the application store.

The second approach is used for native applications that need to use network APIs to access remote hosts, such as to scrape user's smart meter data from an energy company's website. To handle such applications, network communications from a native application are restricted to a specific IP address (or host name). For instance, a native application could be restricted to communicate only with the host name corresponding to the energy company's server. Moreover, read or write access from a native application to the database can also be restricted. In the case of the energy data scraping application, it can be restricted to only write the scraped data, with no privileges to read other types of sensor data. As we show in Section 3.3, these restrictions on data access are easy to accomplish in our prototype implementation. Similarly, we can also restrict the set of web services that a native application can access.

Certified native applications are suitable for data mining, analytics, visualization, appliance control and home automation. Native applications can also obtain users' smart meter data from energy companies and store it in the datastore, making them ideal for data consolidation.

Cloud-Based Applications (CBAs)

Unlike native applications, cloud-based applications are hosted using third parties' hosting services. The main purpose of a cloud-based application is to allow sharing and comparison of sensor data between different VHome instances. ACMs provide fine-grained

access control over time-series sensor data, thus allowing users to choose which part(s) of her data are shared with a CBA, and the time instant(s) at which it is shared, for example, periodically. The challenge lies in preserving privacy while allowing meaningful computations and comparisons. While certified native applications can be given access to raw data, data given to a CBA must be pre-processed using techniques that ensure that privacy is preserved. These actions are implemented by and configured using the PPMs (as explained above). Similar to native applications, CBAs can be published on the application store, and VHome owners can provide CBAs with their VHome URL and explicit authorization to read required parts of their data.

3.2.4 User Interfaces

The gateway, the VHome's web services (WS), and cloud-based applications all allow user interaction. These interactions are mediated using user interfaces implemented on a user device, such as a web browser, a smartphone application, or other mediums like e-mail or SMS. User interfaces serve to simplify the management and control of a VHome instance. Examples include user interfaces that are used to install native applications, to configure the permissions granted to a CBA, and to control appliances in the home.

3.3 Implementation Details

This section presents details of different components of our prototype implementation of the proposed architecture.

3.3.1 Gateway

We implement a software-based gateway using the Microsoft HomeOS [74,75] platform. As described in Section 2.1.2, HomeOS is a .NET based platform designed to provide centralized control of devices in the home such as light switches, thermostats, cameras, and televisions. HomeOS provides developers with homogeneous programming abstractions to orchestrate such devices. We use these features for monitoring and controlling appliances and to enable the uploading of data to the VHome. Figure 2.2 shows an overview of HomeOS. HomeOS is comprised of software modules called *drivers* that communicate with devices to allow higher level application modules to actuate the devices. In addition, a *platform* module manages and coordinates all other modules. To

instantiate our gateway, we extend HomeOS by enriching it with a few additional modules, described next.

Driver Modules

Each driver module monitors and controls an individual appliance using a sensor. We implement driver modules for the Aeon appliance sensor [2] and the CC Envi [8] power and temperature sensor. The Aeon sensor is installed in-series with an appliance and communicates to the gateway over Z-Wave [39]. The module is invoked by the *coordinator module* (described next) for polling data or controlling the sensor, and transmits the respective Z-Wave frames to the desired sensor. The Envi sensor measures the active power from a home every 6 seconds using a Hall-effect transducer. The transducer can be clipped around the split-phase wires of an appliance, or to those at a home's main electricity supply. Measurements are transmitted wirelessly to the Envi console which is connected via a USB cable to the gateway machine (an inexpensive netbook in our prototype). The netbook caches data on local disk and transmits it periodically (e.g., daily) to a VHome. Figure 3.2 shows the netbook running the gateway software, and the Envi device.

Communication Module

This module provides communication between the VHome and the gateway. We use XMPP [174], the protocol underlying the Jabber chat client, as our transport protocol because it uses a simple RPC mechanism that is secure, extensible, and provides real-time communication. Most importantly, XMPP ensures seamless communication from the VHome to the gateway despite the presence of NAT devices, firewalls, and intermittent disconnections at the home gateway.

Coordinator Module

This module records and processes energy data generated by the sensors' driver modules and caches it temporarily on the gateway. Periodic data uploads are received by the VHome's web services and are robust to intermittent losses of connectivity. To facilitate coordination of sensor data and control between the gateway and the VHome, each sensor is assigned a *class ID* and an *object ID*. Sensors of the same type are assigned the same class ID, but distinct object IDs. This allows a VHome instance to identify each sensor

Figure 3.2: Gateway and Envi device.

using the {class ID, object ID} tuple. The coordinator module uses the communication module to listen for commands from the VHome (to control sensors). For example, if Aeon sensors have the class ID 1, and the one interfaced with the electric heater has an object ID 2, then the VHome issues the following command in XML to order the gateway to turn it off.

```
<setStatus classID=1 objectID=2>
  <power>0.0</power>
</setStatus>
```

The gateway performs the required action and responds with the sensor's new status as an acknowledgement. Other actions, for example, dimming lights and managing AC temperature setpoints, are performed in a similar fashion.

3.3.2 VHome

In our prototype, all VHome sub-components – the APIs, Web Services (WS), Access Control Mechanisms (ACMs), Privacy Protection Mechanisms (PPMs), and Native Applications, are implemented as separate Java Web Applications (or *webapps*) using the

Java API for Representational State Transfer (REST) [149] Web Services (JAX-RS framework) [57]. As a result, they can be deployed in a Java Web Container. We use Apache Tomcat as the web container, which is deployed on a virtual machine using the Amazon EC2 cloud [4]. For data storage, we use MySQL as the relational datastore running on the virtual machine's local disk. Our choice of Java was motivated by our desire to ensure VHome portability across virtual machine OSes.

Similar to the data format used for sensors (described in Section 3.3.1), sensor data is organized into classes where each class describes a type of data stream and has a unique class ID, a class name (e.g., heating), a descriptor (e.g., space heaters in the home), and a rating (e.g., 500 W). Data streams can either originate from the sensors at home or can be external (e.g., smart meter readings from an energy company's website). Particular streams of a class are identified as objects using a unique object ID within their class, and have an object name (e.g., master bedroom heater), a descriptor (e.g., installed on 01/01/2011) and a granularity (e.g., 60, indicating data is produced every 60 seconds).

Privacy Protection Mechanisms (PPMs) are also implemented as webapps and access a data stream or streams via APIs. They can create new privacy-preserving streams, which can then be shared with cloud-based applications (CBAs). For instance, we implement *aggregation* as an example PPM where energy consumption time-series data, such as that produced every few seconds, is aggregated to compute daily or weekly consumption values which are less revealing in nature.

The webapp backend for the VHome APIs implements them as a set of TLS-Secure REST [149] URIs, which are used by native and cloud-based applications to access data, control sensors and actators. Table 3.1 provides a brief overview of the API that native and cloud-based applications invoke using HTTP GET or POST requests. Results are returned using JavaScript Object Notation (JSON). Applications can add or modify data streams subject to the Access Control Mechanisms (ACMs). Applications can potentially compute a hash of a data stream (e.g., MD5) and sign it (e.g., using user's private key) to ensure data integrity.

The ACM webapp regulates applications' access to all or a subset of APIs, configured using the VHome Web Services (WS). By default all APIs are regulated and therefore, require a valid access token to return results. The ACM webapp implements OAuth 2.0 [97], a token based authentication and authorization standard for securing API access. It uses the VHome datastore to store data concerning access controls (e.g., tokens, access lists, and more), and this data is only accessible to the ACM webapp.

Figure 3.3 illustrates this access process for a cloud-based application (CBA) hosted as a web portal. To access any API, the CBA is first required to obtain a *one time authorization*

grant from the ACM webapp by providing its identity (identifier, name, or host-URL) and a list of APIs that it requires access to and the parameters to the APIs. For instance, a CBA requiring access to the bedroom space heater consumption data (e.g., with class ID 1 and object ID 2) from January to March 2012 would request access to the data stream API as:

```
https://<VHome URL>/GetStream/classID/1/objectID/2/TS/1325394000/1333252799
```

where TS indicates time-series data, and 1325394000 and 1333252799 are the epoch timestamps at 01-01-2012 00:00:00 and 31-03-2012 23:59:59, respectively.

This allows restricting the scope of data access to certain data streams and/or certain segments of a stream's time-series defined using timestamp and/or data values. The ACM webapp then redirects the user to the Web Services (WS) webapp so as to authenticate the user as the VHome owner. After authentication, the scope and nature of the requested access is described to the user, and her authorization for the access is requested. The WS implements this as a simple notification in a web-browser, which can be relayed to other remote UIs such as email, SMS, or mobile application notification. An example of such a notification from an application named "EXAMPLE" is:

> The application named EXAMPLE is requesting
> access to bedroom space heater data
> for Jan 1 to Mar 31, 2012.
> Allow or Deny ?

The CBA is then required to exchange the one time authorization grant (before it expires) to obtain an *access token* and an (optional) *refresh token*. The access token permits a CBA to use the corresponding APIs until the token expires, after which a new access token must be obtained using the refresh token. All tokens are valid for periods configured by the user. By matching the CBA's credentials (e.g., URL) to those registered while issuing the authorization grant, the ACM validates each API access and prevents use of stolen access tokens. Further, if at any point the user decides to revoke (or pause) a CBA's access to data, she can simply revoke the access token and the refresh token for that CBA.

Our prototype implements the authorization grant, access token and refresh tokens in the form of randomized 128-bit MD5 codes, where the webapp maintains a lookup table that stores their scope and expiry times. Avoiding the encapsulation of scope and

duration within the token circumvents token processing overhead for each API access. The authorization grant and access token issuing endpoints are published as GET/POST URIs by the VHome, and use JSON for token and error-message exchanges with CBAs. Our prototype implementation assumes one user per VHome. We defer the management of multiple users per VHome instance to future work.

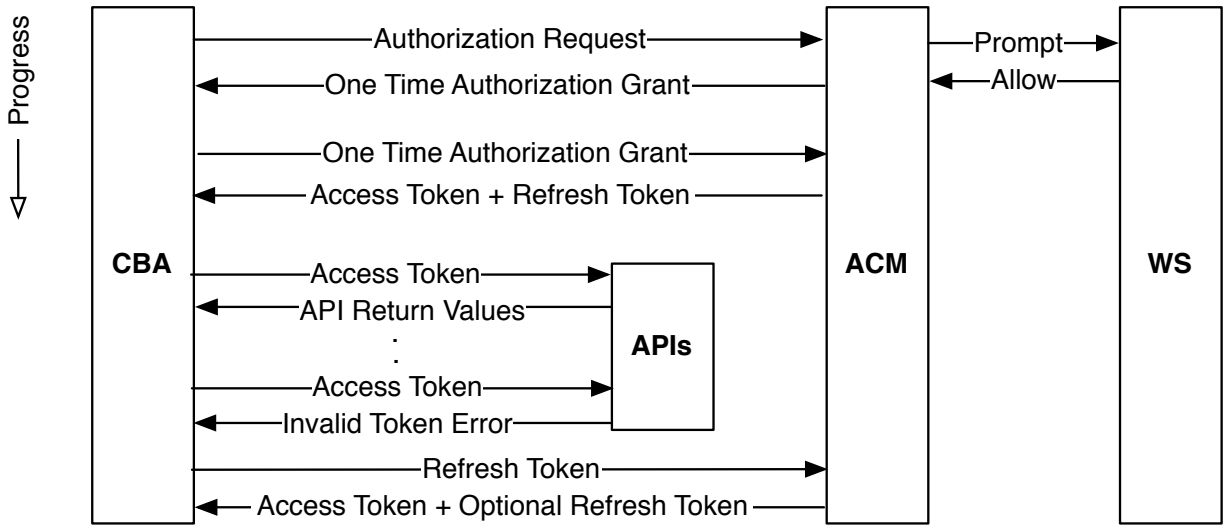


Figure 3.3: API access for a cloud-based application.

The Web Services (WS) webapp implements a number of backend components and allows users to configure them. First, it coordinates periodic data uploads from the gateway over XMPP and symmetrically transmits control commands to the gateway's coordinator module. Second, it provides the user with a control portal to install native applications on the VHome. Native Applications, being JAVA web applications, are then profiled by the WS webapp for use of the JAVA.net interface and for the VHome APIs they require. The user can restrict the applications' ability to read from and write to the database by disallowing or restricting the scope of the APIs. Likewise, users can configure ACM settings such as token formats and expiry periods, and chose which APIs it permits. In a fashion similar to native applications, certified PPMs can be added to the VHome through this portal which can be run to create additional data streams. Lastly, the WS webapp allows users to purge native applications or data, and revoke access tokens of any CBA.

Our prototype VHome implementation is open source and can be found at https:

//vhome.codeplex.com while the gateway implementation using HomeOS is hosted at http://homeos.codeplex.com.

Function (regulated by default)	Description
ListAllClasses	Returns all attributes of all classes of data in the VHome DB
ListClass/param/value *param*: class ID, class name or rating	Returns all attributes of classes that match the specified parameter values
ListObject/param1/value1/param2/value2 *param1*: class ID, class name or rating *param2*: object ID, object name or granularity	Returns all attributes of objects that match the specified parameter values
AddClass/className/x/descriptor/y/rating/z	Adds class with name x, descriptor y, and rating z
AddObject/classID/w/objectName/x/descriptor/y/granularity/x	Adds object with given class ID w, object name x, descriptor y, and granularity z
AddStream/classID/x/objectID/y/	Adds time-series data to the given data stream with class ID x and object ID y
GetStream/classID/x/objectID/y/	Returns the complete time-series data stream with class ID x and object ID y
GetStream/classID/x/objectID/y/TS/t_1/t_2	Returns the time-series with class ID x and object ID y between timestamps t_1 and t_2
GetStream/classID/x/objectID/y/Val/v_1/v_2	Returns the time-series with class ID x and object ID y between data values (or Val) v_1 and v_2
GetStatus/classID/x/objectID/y	Returns the current power consumption of device with class ID x and object ID y
SetStatus/classID/x/objectID/y/status/p	Sets the power consumption of device with class ID x and object ID y to p

Table 3.1: VHome API used by applications to access data.

3.3.3 Sample Applications

We have created a sample application store as a web portal where users can browse for native applications. It transfers the selected applications' executables to the VHome's web services (WS) which installs them on the VHome, and can then be accessed using remote UIs. We now describe four applications that we have built using our system which, without our architecture, cannot be implemented in a privacy-preserving form.

Data Scraper

This application obtains users' smart meter data from the energy company. It is implemented using the JAVA DOM interface as a VHome native application. Our prototype

application runs on the VHome, scrapes data from an energy company's web portal and stores it in the datastore. At installation time, the application requests users to provide it with their access credentials to access the energy company's portal (Waterloo North Hydro [36] in our prototype). The application also allows users to set automated periodic data scraping actions to ensure that data is obtained before it is discarded by the utility company's portal (e.g, after three months) and the user is relieved of manually retrieving the data. The application allows the data to be retained by the user even after it is no longer available on the energy company's portal. The data is stored as a data stream in the *Smart Meter* class which can then be accessed by other applications through APIs. Access to historical smart meter data provides opportunities for data analytics. For instance, it allows applications to account for seasonal climate changes when examining the consumption history.

Interactive Monitoring and Control

This native application interfaces with the VHome web services to monitor and control home appliances in real-time from an Internet-enabled device. Note that native applications have no network access and can only be viewed by invoking the trusted VHome webapp container. We implement an Android smartphone application that invokes a VHome native application and provides a smartphone application GUI. This means users can use VHome native applications via web-browsers or with applications installed on their mobile devices such as smartphones and tablets.

Figure 3.4 shows snapshots of different panels in our Android smartphone application. Screenshot-1 (on the left) shows the home monitor, which allows users to view current conditions of the home as reported by the Envi sensor. In addition, the consumption data stored at the VHome is used to compute and display the day's and week's consumption. Screenshot-2 (in the center) shows the control panel which allows users to turn on or turn off different appliances connected to Aeon Z-Wave sensors and displays their current consumption. Further, it allows users to share the amount of energy they conserve by turning off appliances, on social networks such as Facebook and Twitter and to compete with their friends. Screenshot-3 (on the right) shows a past trend of aggregate energy consumption measured using the Envi sensor. This trend data can be used by users to better understand abnormal consumption notifications (described below). Our prototype implements SMS, E-Mail and mobile application notifications. Since the VHome runs on a cloud virtual machine, energy data can be processed and accessed using any connected device with relatively low latency.

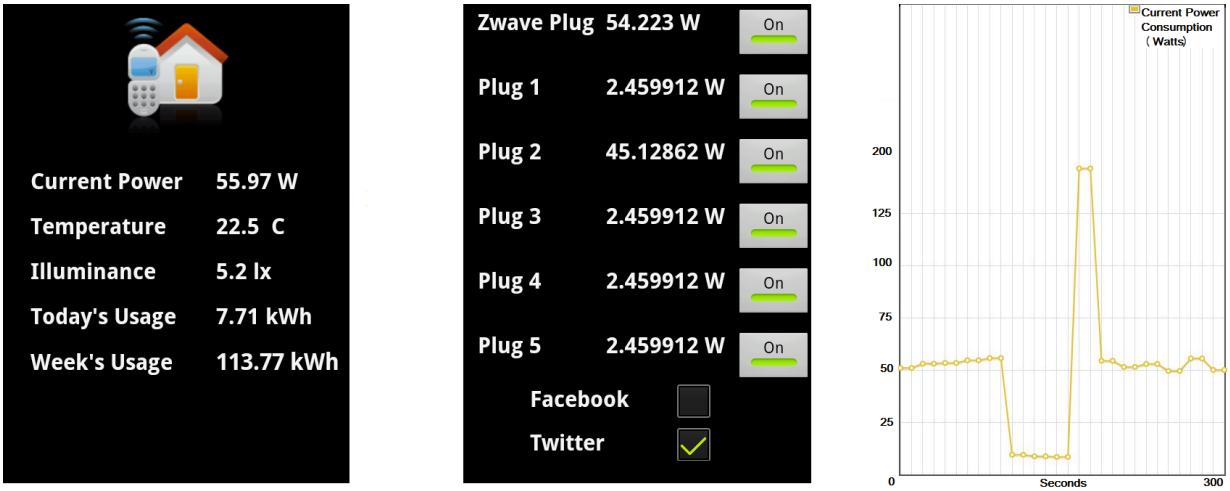


Figure 3.4: Screenshots of our Android smartphone application.

Abnormal Energy Consumption Detection

This VHome native application informs users about abnormalities in their energy consumption. For instance, consider a scenario where users forget to turn off an high-risk appliance such as an oven while they are away. Using the VHome APIs, this application periodically obtains the energy consumption values from the gateway, measured by the Envi sensor. It then compares the values to a predicted value computed using an Auto-Regressive Moving Average model. If the measured value is higher than the predicted value by a threshold (e.g., 1 kW), then the application sends the user a notification message via e-mail, SMS, or the Android smartphone application. The user can then either use the Android application (described above) or reply to the email or SMS, to take appropriate action.

Energy Data Analytics

In many parts of the world the price of electricity depends on the time of the consumption. In Ontario, a day is divided into peak, mid-peak, and off-peak hours, each with different rates [26]. We implement a VHome native application that processes a home's electricity consumption measured using the Envi sensor to determine how much energy is consumed during different hours of the day, its corresponding cost under the pricing scheme, and the total cost. It uses smart meter data obtained and stored by the data

scraper application to verify a user's monthly energy bill. Such simple analytics also provides the user with meaningful insight into her hourly and daily consumption patterns, warns her of potential errors in their energy bills, and can help her to time shift non-critical consumption.

3.4 Evaluation

We compared our architectural approach to other approaches in Section 2.1 (Chapter 2). In this section, we compare our prototype to different existing widely deployed (i.e., non-research) systems for home energy applications. Table 3.2 compares the different systems based on our design goals (Section 1.2).

Commercial software solutions such as Google Powermeter [15] and Microsoft Hohm [24], being centralized web services are scalable but provide a fixed set of analytics with no data consolidation and little data privacy. Both services are now defunct, thus leaving users deprived of their energy data. Energy companies' web portals act similarly and share/discard the smart meter data at their discretion but ensure integrity of that data. The Greenbutton [17] initiative has standardized energy data formats, so that users can access their data and analyze it themselves (denoted "Greenbutton (Self)") by using desktop tools such as Energy Lens [7]. This allows users to choose analytics tools, ensure data integrity, and provides data privacy, but it burdens them with data hosting. Alternatively, users can delegate the data retrieval and maintenance to third parties (denoted "Greenbutton (Third Party)"). Third parties can manage, analyze and host users' energy data, but such unconditional access to raw data provides little data privacy.

Our prototype implementation of the proposed architecture meets the design goals for a user-centric, privacy preserving system for energy data analysis as explained below.

- **Data Consolidation:** VHome native applications, such as the data scraper application (Section 3.3.3), can read data from any data source and store it in the VHome database. This allows users to easily consolidate data from multiple sources by using one native application per data source. User-owned sensors can be directly interfaced with the gateway, in a fashion similar to that described in Section 3.3.1.
- **Data Durability:** Instead of relying on a single computer in the user's home to store data, and relying on the user to back up that data, for example, by making

	Hohm [24], Powermeter [15]	Energy companies web portals [32, 36]	Greenbutton [17] (self) Energy Lens [7]	Greenbutton [17] (third party)	VHome prototype
Data Consolidation			✓	✓	✓
Data Durability					*
Data Integrity		*	✓		*
Data Privacy			✓		✓
Application Extensibility			✓	✓	✓
Application Performance	✓	✓		✓	✓
Scalability	✓	✓		✓	✓

Table 3.2: Comparison of existing solutions for home energy data applications, * denotes a partial solution.

off-site backups, data is stored in the VHome datastore. The VHome prototype is hosted on a user-owned virtual machine (VM), and uses its local disk to warehouse the data. However, different cloud providers provide varied guarantees on the durability of data stored on VMs' local disks [4, 37]. For instance, VMs on Amazon Elastic Compute Cloud (EC2) may lose local disk data in case of hard drive failures. Therefore, our current prototype provides only a partial solution to data durability. In Chapter 4, we address this problem by leveraging commodity cloud storage services for storing sensor data, since they provide stronger data durability guarantees with lower storage cost than VM local disks.

- **Data Integrity:** Native applications can implement mechanisms to ensure the integrity of data stored directly into the VHome datastore, such as sensor data uploaded from the gateway. Example mechanisms include storing signed hashes of data streams. However, our current prototype does not include such native applications. We address this problem in detail in Chapter 4.

  Moreover, in cases of data procured from external sources, such as smart meter data from energy companies' servers, applications have to rely on the respective sources such as energy companies, for preserving integrity of the data. Similarly, our prototype assumes that the VEE and VHome-software providers do not tamper with data in the VHome datastore.

- **Data Privacy:** The current prototype uses several mechanisms to ensure data privacy. First, data within a VHome instance is not accessible by entities outside the VHome, eliminating many types of privacy violations. Privacy leakage from na-

tive applications is prevented by certifying native applications, by checking Java byte code submitted to the application store to ensure that they are either not using network APIs, or when they need to, are only communicating with the specified hosts. The details of this process are described in Section 3.2.3. Privacy leakage from cloud-based applications is mitigated, to some extent, by aggregation or obfuscation and other privacy preserving mechanisms (PPMs). Note that, these protections of data privacy assume that the VEE and VHome-software providers are trusted.

- **Application Extensibility:** Users can freely choose to extend their set of VHome native applications by installing them from the application store. Users can also chose to transfer their data to other cloud-based applications for analytics and other services.

- **Application Performance:** Personal VEEs in the cloud provide VHome applications with more computational resources such as memory, CPU time, than are available on a typical user's home machine. Moreover, hosting data and applications in the cloud minimizes remote access latencies as compared to typical home access links which have lower bandwidths.

- **Scalability:** In contrast with home-based computers, cloud-based personal VEEs allow for easy scaling of both data set sizes and computation, by suitable provisioning and re-sizing of the VEEs.

Thus, our prototype addresses all of our design goals, and demonstrates the feasibility of our proposed personal VEE architecture using existing hardware, software, and cloud infrastructure.

3.5 Discussion

Smart Meter Data

In some sense, ensuring that smart meter data remains private is moot, because energy companies today collect this data and share it arbitrarily with third parties of their choice (e.g., Google PowerMeter and Microsoft Hohm), without seeking users' explicit permission. However, this situation is likely to change in the future due to two reasons. First, we anticipate that many jurisdictions, following the lead set by the province of Ontario

(in Canada), will place severe restrictions on the sharing of smart meter data with third parties, thereby freezing innovation in data analytics and customized recommendation applications. Although this is being countered by proposals such as the Green Button initiative [17], which release smart meter data back to users, users are not capable of performing their own data analytics, and are loathe to share this data with third parties due to privacy concerns. Second, besides smart meter data, users are likely to generate many other equally private data streams including health-monitoring data, especially with the widespread adoption of health-tracking devices such as Fitbit [14]. Our prototype implementation can be extended to applications that analyze and process other such data streams, thus balancing data privacy and application innovation in those domains.

An alternative approach for handling smart meter data is one where a home's data is transferred directly from the smart meter to a homeowner's personal VEE. A VHome native application can then use the data to compute the monthly or yearly bill which is relayed to the energy company. Existing work on zero-knowledge proofs [136] and trusted computing platforms [130], may be leveraged to guarantee the authenticity and validity of the bill to the energy company, which has no access to the raw privacy-sensitive data.

Incentives

Our current prototype caters to home energy data, and involves five stakeholders– energy companies (that collect smart meter data), users, personal VEE providers, VHome software providers, and application developers. We now explain the incentives for each stakeholder to participate in the system.

- **Energy Companies:** Energy companies are under great pressure from legislatures to release smart meter data to the users, as evidenced by the Green Button initiative [17]. They also benefit from users' adoption of energy conservation applications, in that it reduces their need for costly generation capacity upgrades. For instance, energy companies often have to provision additional power plants to serve their peak electricity demands, even though such demand levels occur rarely.
- **Users:** Users are increasingly becoming aware of the costs of the world's rampant energy consumption. In some cases users are motivated to better understand and reduce their consumption to limit their carbon emissions, and its disastrous consequences such as global warming and climate change. In other cases, users are motivated to reduce their energy bills. Users currently lack the infrastructure and tools to understand how they can achieve these reductions without giving up their data privacy.

- **Personal VEE providers:** VEE providers are paid by users for hosting and maintaining their services, and thus have a monetary incentive for participation.
- **VHome software providers:** We believe that VHome software providers can be compensated for their development and maintenance efforts in two ways. First, some user may wish to pay for purchasing VHome software, so that they can receive intelligent recommendations for their electricity use. This is akin to those users who pay a monthly fee for services such as DropBox. Second, vendors of energy-efficient appliances could subsidize the cost of VHome software, because applications' recommendations for the use of their products, such as energy-efficient air conditioners, washing machines and LED lights, will lead to increased product sales.
- **Application developers:** Application developers receive a mass audience for their applications, and can choose to either sell them directly, or use builtin advertisements; similar to applications on application stores such as the Apple App Store, and Google Play. Some applications may also be commissioned by equipment vendors, as discussed above.

Key Aspects

Our proposed architectural approach may appear to be obvious, merging cloud-hosted data and execution environments with sensor data streams, an approach already implemented by data management systems such as Pachube [29]. However, there are three aspects of our work that are not obvious. First, we show how to use virtualized execution environments, in conjunction with an object-level framework, to provide a practical solution for the seemingly conflicting requirements of ensuring data privacy while fostering application development. Second, our approach enables the development of an ecosystem of energy management applications in much the same way as Google Play and the Apple App Store provide an ecosystem for the development of smartphone, tablet, and desktop applications. Third, our approach is diametrically opposed to the service-provider centric view that is widely prevalent in the Smart Grid community, and commonly used by energy companies. Instead of designing an architecture that caters to the needs of energy companies, our approach places control firmly in the hands of the users.

Limitations

Our prototype is limited to dealing with sensor generated time-series data and does not support other potential forms of user data, such as photos, videos, or document files.

Secondly, since our approach provides users with greater control over their data than existing approaches, hence the user is faced with an increased number of decisions. However, such cognitive overload may be eased by learning users' preferences from user studies to help make decision-making simple and intuitive for the user.

Our prototype does not include mechanisms for ensuring integrity of data streams. Although, as explained in Section 3.4, these mechanisms as fully realizable in our architecture. Moreover, such mechanisms are particularly important if commodity cloud storage services are used for long-term data storage. Similarly, our prototype uses an existing relational database system (MySQL) to store the sensor data. This means that some applications' latencies may be impacted by the database's retrieval latencies. However, it may be possible to improve retrieval latencies by leveraging the nature of data queries that are typically issued by applications. We explore these possibilities in Chapter 4.

Lastly, our approach also requires a trusted certification mechanism to certify applications and requires VHome software and personal VEE providers to be non malicious.

3.6 Chapter Summary

The work in this chapter investigates the feasibility of the personal VEE approach for enabling an ecosystem of privacy-preserving data-driven applications. To do so, we choose home energy data as an example use case because of its widespread potential impact. We build a prototype implementation of our proposed architecture that helps users manage home energy data and its applications. We identify, design, and implement the different components of the architecture. We also demonstrate how existing work, both in energy data privacy and home device management can be leveraged to enrich some architectural components, such as an in-home gateway and privacy protection mechanisms that transform sensor data to be less revealing. Our prototype consolidates and stores users' home energy data, while exposing it to third party applications while ensuring that users have complete control over their data. It addresses all of our design goals, and demonstrates the feasibility of our personal VEE architecture using existing hardware, software, and cloud infrastructure. We conclude the chapter by presenting a discussion of smart meter data with regards to our architecture. We also outline the

incentives for different stakeholders in our prototype, and discuss the key insights that underlie our work, its current limitations and possible improvements.

Chapter 4

A Storage System for Sensor Data

4.1 Introduction

In our prototype implementation of the personal VEE architecture (described in Chapter 3) we provided native and cloud-based VHome applications with RESTful APIs to read and write sensor data. We implemented a few example applications to demonstrate the use of these APIs. However, our prototype suffers from two main drawbacks.

First, many applications require richer data manipulation capabilities than those supported by our prototype data storage system. For instance, PreHeat [161] is an application that infers home occupancy by processing in-home motion sensor readings, and uses historical inferred occupancy values from specific time windows to predict future occupancy which is used to control home heating to reduce home energy consumption. Similarly, Digital Neighbourhood Watch (DNW) [56, 68] is an application that stores information about physical objects seen by in-home cameras, and shares this information with neighbouring homes when requested, allowing users to gather image and video evidence of criminal activities in the neighbourhood. We describe these applications in detail in Section 4.2.1. Building these applications as native or cloud-based VHome applications will require developers to design and implement suitable data retrieval and sharing mechanisms, thus increasing development effort.

The second drawback of our prototype storage system is that it only uses local VEE storage. However, users and applications may also want to use commodity cloud storage services such as Windows Azure [37] or Amazon S3 [4] to lower storage costs or obtain better reliability. This requires application developers to not only design mecha-

nisms to store sensor data using these services but also to provision additional mechanisms to prevent storage providers from accessing users' private data.

We aim to simplify the management of data from in-home sensors by designing a suitable data management system that replaces our prototype storage system. To do so, we survey existing and proposed home applications and determine the key requirements for such a system. We now briefly summarize these requirements and describe how they are derived in detail in Section 4.2.

- The system should support time series sensor data and should allow arbitrary tags to be assigned to values. This is because tags provide applications with a flexible way to assign application-specific semantics to time series values. For instance, the DNW application may use "car" as a tag for a given data record.
- The system should enable sharing of data across VHome instances because many applications need to access data from multiple homes. For instance, an application that correlates energy data across homes to provide users with a comparison of their energy consumption with other similar homes in the neighbourhood.
- The system should support cloud storage providers and should allow applications to freely specify storage providers. This is because applications are in the best position to prioritize storage metrics such as cost, reliability, location, and latency.
- When using cloud storage, the system should protect data against eavesdroppers in the cloud or in the network because of the privacy-sensitive nature of sensor data.

Unfortunately, existing systems (described in detail in Section 2.2) do not meet all of these requirements.

Therefore, we design and implement *Bolt*, a data management system that meets these requirements to provide storage, retrieval, and sharing of sensor data with low resource overheads. Bolt abstracts data as a stream of $(timestamp, tag, value)$ tuples, and builds indices on these tuples to enable applications to issue data queries based on timestamp and tag values. It uses cloud storage as a seamless extension of local storage, and allows applications to prioritize storage across providers. Cloud storage is also used to facilitate the sharing of data across VHome instances. Bolt encrypts sensor data before uploading it to cloud storage services, and maintains metadata on a trusted server (a native VHome application) to manage data sharing and provide data integrity guarantees.

Our key contributions in this chapter are:

- A formulation of data management requirements of in-home data-driven applications.
- The design and implementation of Bolt, a system for the storage, retrieval, and sharing of in-home sensor data that meets the design requirements.
- A performance evaluation of Bolt using three sample applications showing that Bolt has up to 40 times lower data retrieval time and incurs 3–5 times less storage space than OpenTSDB [27].[1]

This work has been published in the proceedings of USENIX NSDI 2014 [93].

The remainder of the chapter is structured as follows. We survey existing applications and formulate the design requirements in Section 4.2. Section 4.3 describes the data guarantees Bolt provides and gives an overview of the key elements of Bolt's design. We describe the design in detail in Section 4.4 and our current Bolt implementation in Section 4.5. We evaluate Bolt in Section 4.6 and present a discussion of potential extensions of Bolt in Section 4.7. We conclude with a summary of the chapter in Section 4.8.

4.2 Design Requirements

To determine the design requirements for our storage system, we surveyed several applications that process in-home sensor data, including example applications described in Chapter 3. We first briefly describe three such applications chosen because of their diversity in the type of data they manipulate, their data access patterns, and their potential implementations as native and cloud-based VHome applications. We then formulate the design requirements in Section 4.2.2.

4.2.1 Example Applications

PreHeat

PreHeat [161] correlates readings from motion sensors in a home to infer users' home occupancy. It records the occupancy values, and uses past occupancy patterns to predict future occupancy, which it uses to turn a home's heating system on or off to reduce the

[1] A popular time series data management system, discussed in Section 2.2.3.

amount of energy consumed for heating the home. PreHeat divides a day into 15-minute time slots (i.e., 96 slots/day), and records the occupancy value at the end of a slot: 1 if the home was occupied during the preceding slot, and 0 otherwise. At the start of each slot, PreHeat predicts the slot's occupancy value, using the slot occupancy values for past slots on the same day and corresponding slots on previous days. For instance, for predicting the occupancy of the nth slot on the dth day, PreHeat first retrieves occupancy values for slots $1 \cdots (n-1)$ on the dth day. This is called the *partial occupancy vector (denoted as POV_d^n)*. In addition, PreHeat also uses POV_{d-1}^n, POV_{d-2}^n, $\ldots$, POV_1^n. From this set of past POVs, PreHeat selects K POVs which have the least Hamming distance to POV_d^n. An average of these top K POVs is computed, then it is compared with a threshold value to obtain the predicted occupancy value and the heating system is turned on or off accordingly. Since PreHeat only requires sensor data from a single home, it can be implemented as a native VHome application in our personal VEE architecture.

Energy Data Analytics (EDA)

Recall that the sample EDA application we considered in Chapter 3 processed energy sensor data values to determine how much energy is consumed during different hours of a day and the energy costs under a time-of-day based pricing scheme. However, recent work has proposed improved data analysis techniques for energy data. For instance, techniques that identify the energy consumption of different appliances, user activities, and energy wastage [54, 83, 119, 140].

A specific EDA application that we consider is a cloud-based application, for example, hosted by an energy company, that presents an analysis of the monthly energy consumption to users [54]. It disaggregates hourly home energy consumption values, such as smart meter readings, into different categories which include: baseline consumption, user activity driven consumption, consumption of heating or cooling systems, and others. For each home, the variation in consumption level is analyzed as a function of ambient temperature by computing the median, 10th, and 90th-percentile home energy consumption levels for each temperature value. These values are then compared with other homes in a neighbourhood or city, and are reported to the user with appropriate graphics.

Digital Neighbourhood Watch (DNW)

DNW helps neighbors jointly detect suspicious activities, for example, an unknown car cruising the neighbourhood, by sharing security camera images [56, 68]. It can imple-

mented as a combination of a native and cloud-based VHome applications. A DNW instance in each VHome monitors the footage from security cameras in the home. When it detects a moving object, it stores the object's video clip and generates its summary information, such as:

```
Time: 15:00 PDT, 23rd September, 2014
ID: 001
Type: human
Entry Area: 2
Exit Area: 1
Feature Vector :{114, 117, ... , 22}.
```

This summary includes the inferred object type, its location, and its feature vector, which is a compact representation of its visual information. The example above shows the feature vector for a sample human subject.

When a user deems a current or past object suspicious, the DNW instance in her VHome queries neighbouring homes' instances to query if they have also seen the object around the same time. A DNW "coordinating endpoint", implemented as a cloud-based application, could be used to perform such queries. In response to such a query, each neighbour home instance extracts all objects that it saw in a time window (e.g., an hour) around the time specified in the query. If the feature vector of an object is similar to the one in the query, it responds positively and optionally shares the video clip of the matching object. Responses from all the neighbours allow the original instance to determine how the object moved around in the neighborhood and if its activity is suspicious.

Other Applications

Other applications we surveyed include Digiswitch [59], which allows elders and their distant caregivers to share real time home activity information. We also surveyed commercial systems such as Kevo (which interface with in-home connected door locks) and other home energy applications that we implemented in Chapter 3 including: abnormal consumption detection, interactive monitoring and control, and smart meter data scraper. The requirements, which we describe next, also cover those placed by these applications.

4.2.2 Data Management Requirements

We formulate the requirements of the applications (described above) into four categories.

1. Support Time-series, Tagged Data

Most in-home sensor applications generate time-series data and retrieve it based on time windows. For instance, as described above in Section 4.2.1, PreHeat stores continuous home occupancy values, and retrieves them based on multiple time windows (called POVs). Similarly, EDA accesses hourly energy consumption values, and object summaries in DNW are also time-series data. Moreover, applications also sometimes tag the data, so that it can be queried at a later time using application-specific tags. For instance, object type "car" is a possible tag in DNW and "home heating consumption" is a possible tag in EDA. Applications can specify such tags to retrieve the required values.

We also observe other commonalities in the data manipulation patterns of applications. First, time-series data in these scenarios has a *single* writer. Second, writers always generate new data and do not perform random-access updates or random record deletions. Third, readers typically fetch multiple proximate records by issuing temporal range queries for sliding or growing time windows. Note, however, that traditional databases incorporate transactions, concurrency control, and recovery protocols that incur unwanted resource and latency overheads [162]. Unfortunately, filesystem storage offers inadequate query interfaces for such applications.

2. Support Policy-Driven Storage

Different types of time series data have different storage requirements for location, access latency, cost, and reliability. A DNW application may record in-home images locally (e.g., on a personal VEE's local disk) and can delete them once it has extracted and stored the summaries of objects in them. The application may want to store these on a remote server to enable correlation with images captured by neighbouring homes, for example, to detect a suspicious car moving in the neighbourhood. Alternatively, the raw images may be stored on cheaper archival cloud storage. We believe that applications are in the best position to prioritize storage metrics and should be able to specify the policies that govern the storage of data (that they generate) across providers.

3. Ensure Data Confidentiality and Integrity

Applications may also use remote storage infrastructure to simplify data management and sharing, improve reliability or lower cost, but may not trust the storage services with maintaining the confidentiality or integrity of data. Data processed by applications may reveal immensely private information about the user: occupancy and energy data in PreHeat and EDA respectively, can reveal when residents are away, which can be exploited by attackers. Therefore, a data management system for in-home sensor data should guarantee the confidentiality and integrity of stored data. The system should also support efficient changes in access policies, without requiring, for instance, re-encryption of large amounts of data.

4. Efficiently Share Data Across Homes

Many applications require access to data from multiple homes. In our personal VEE architecture, these are either VHome native applications that need to share data with other VHome instances or are cloud-based applications which correlate data from multiple VHome instances. Both DNW and EDA are examples of such applications, which share object summaries and energy consumption values across homes respectively. Existing data storage and sharing services, such as Dropbox or OneDrive, can simplify cross-home sharing but they will unnecessarily synchronize large quantities of data. Applications may want to access only part of the data produced by a sensor. For instance, in DNW, it would be wasteful to access an entire day's video data if the search for suspicious objects needs to be conducted only over a time window of few minutes.

Designing a storage system such that it meets all the requirements described above, presents many design choices. For instance, storing all data locally facilitates confidentiality but inhibits efficient sharing and reliable storage in the cloud. Similarly, storing data in the cloud provides reliable storage and sharing but untrusted storage servers can compromise confidentiality. Sharing data by synchronizing large amounts of data is inefficient, whereas naïvely storing encrypted data on untrusted cloud storage servers inhibits efficient sharing.

As described in detail in Section 2.2, existing systems either provide inefficient sharing and query abstractions for time-series data [80, 85, 126], assume partial or complete trust on the storage servers [127], or store data locally while ignoring application storage policies [87]. In the following sections we describe the design of Bolt, which meets the requirements discussed above.

4.3 Design Overview

Bolt provides applications with a *stream* abstraction, where each stream is a collection of records, and each record has a timestamp and one or more tag-value pairs, that is, <timestamp, <tag1,value1>, [<tag2, value2>, ...]>. Streams are uniquely identified by the three-tuple: <VHomeID, AppID, StreamID>. Bolt allows retrieval and filtering of streams' records using time ranges and tags. We first explain our design assumptions and Bolt's data guarantees followed by a description of key design elements in Section 4.3.2. Highlighting the key design elements enables us to describe the design in detail in Section 4.4.

4.3.1 Security Assumptions and Guarantees

Bolt does not trust the cloud storage servers to maintain data confidentiality or integrity. It assumes that the storage infrastructure is capable of performing unauthorized reads or modifications to stream records and can return old data when queried. By building on top of this untrusted storage infrastructure Bolt provides the following three data security guarantees:

1. **Confidentiality:** Data in a stream can be read only by an application to which the owner, that is, the writer, grants access. Once the owner revokes access, the reader cannot access data stored *after revocation*.

2. **Tamper Evidence:** Readers can detect if data has been tampered with by anyone other than the owner. However, Bolt does not defend against denial-of-service attacks, for example, where a storage server deletes all data or rejects all read requests.

3. **Freshness:** Readers can detect if the storage server returns stale data, that is, data is older than a given owner-configurable time window.

4.3.2 Key Techniques

We describe the four main techniques that allow Bolt to meet these design requirements.

Chunking

Bolt stores data records in a log per stream called the *DataLog*, which enables low-latency append-only writes. Streams have an index into the DataLog to support efficient lookups, filtering on tags, and temporal range and sampling queries. A contiguous sequence of records within a log constitutes a *chunk*. A chunk is the basic unit of transfer for storage and retrieval. Data writers upload chunks instead of individual records. Bolt compresses chunks before uploading them, which lowers transfer time and storage space required. Readers also fetch data at the granularity of chunks. Although, this may fetch more records than are needed for answering a given query, the resulting inefficiency is partially mitigated by the fact that applications, such as the ones surveyed in Section 4.2.1, are often interested in multiple successive queries rather than a single query. Delay incurred for common queries with temporal locality is improved by fetching chunks instead of individual records because it avoids additional round trip delays.

Note that typical sensor data, when packed into chunks, has high compression ratios that lowers the fraction of bytes transferred when fetching chunks for serving data reads. Chunks can be compressed using existing compression techniques such as GZip and delta encoding of timestamps and values of records within a chunk, which will further improve storage and transfer efficiency. These techniques do not have a significant impact on retrieval time because chunks can be uncompressed and decoded, in parallel, using a pipeline.

Separation of Index and Data

Bolt maintains an index for each stream. The index stores information about the location of different records stored in the stream's log. When answering queries for data stored remotely, Bolt first fetches and stores the stream index on local disk. This separation of the index and DataLog, enables two key properties.

First, when answering read queries for data stored remotely, the index (fetched and stored locally) can be used to determine the chunks that should be fetched from remote servers. A dedicated computation endpoint, such as a query processing engine hosted in the cloud, is therefore not required, thus reducing storage and retrieval costs. This allows Bolt to use existing storage servers that only provide `get` and `put` APIs.

Second, this separation allows Bolt to relax its trust assumptions for storage servers, supporting untrusted cloud providers without compromising data confidentiality by encrypting data. The data can be encrypted before storing and decrypted after retrievals,

while the storage provider does not need to support any data semantics. Using untrusted cloud providers is challenging if the provider is expected to perform index lookups on the data.

Segmentation

Since applications only append new data and do not perform random writes, stream DataLogs can grow very large over time. Bolt allows archiving of contiguous portions of a stream, that we call *segments*, while still allowing efficient querying. The storage location of each segment can be configured independently, enabling streams to be stored across multiple storage providers. Hence, streams may be stored either locally, remotely on untrusted servers, replicated for reliability, or striped across multiple storage providers for cost effectiveness. This configurability allows applications to prioritize their storage requirements of space, latency, cost, and reliability. Bolt currently supports local disk, Windows Azure storage, and Amazon S3 as storage providers.

Decentralized Access Control and Signed Hashes

To maintain confidentiality when using untrusted storage servers, Bolt encrypts the stream with a secret key generated by the owner, that is, the writer. Our design supports encryption of both the index and data, but by default we do not encrypt indices for efficiency[2], though in this configuration some information may be leaked through data stored in indices.

We use lazy revocation [105] for reducing computation overhead of cryptographic operations. Lazy revocation prevents evicted readers from accessing content stored after revocation, because any content stored before revocation may have already been accessed and cached by such readers. From the different well-known key management schemes that support lazy revocation, we use hash-based key regression [84] for its simplicity and efficiency. It enables the owner to share only the most recent key with authorized readers, based on which the readers can derive all previous keys to decrypt any content encrypted using those keys.

We use a trusted *metadata server* to store and distribute keys. It runs on a user's personal VEE. Once an application has opened a stream, all its subsequent reads and writes

[2]Index decryption and encryption is a one time cost paid at the start and end of a session of queries respectively (called stream `open` and `close`), and is proportional to the size of the index.

occur directly between the storage server and the application. This prevents the metadata server from bottlenecking read or write operations. In addition to keys for encrypted data streams, the metadata server also stores additional per-stream metadata, which we describe in detail in Section 4.4.4.

Encryption provides confidentiality of data. However, a remote untrusted server storing part of a stream may modify data records or could return old copies of the data. Therefore, we incorporate data integrity and freshness checks into each stream. To facilitate integrity checks on data, the writer generates a hash of stream contents, which is verified by the readers. To enable freshness checks, similar to SFSRO [85] and SiRiUS [89], we include a freshness time window as a part of a stream's integrity metadata (denoted MD_{int}). This time window denotes the time up to which a stream's data can be deemed fresh; it is based on the periodicity with which writers expect to generate new data, since typical writers periodically append new values to streams. Writers update and sign this time window periodically, which is used by readers to verify records when they open a stream for reading.

4.4 Bolt Design

This section describes the design of Bolt in greater detail.

4.4.1 APIs

Table 4.1 shows the different stream APIs offered by Bolt. Applications, identified by the tuple (VHomeID, AppID), are the principals that read and write data. Table 4.2 shows the stream properties that applications specify when performing a stream `create` or a stream `open` operation. These properties define the stream's type, storage location, encryption, and sharing requirements. The stream type can be a *ValueStream*, which can be used to store relatively small data values such as temperature readings, or a *FileStream*, which is used for relatively large data values such as images or videos. As we detail below, the two types of streams organize data in different ways on disk.

Each stream has one writer (owner) and one or more readers. Writers add time-tag-value records to the stream using the `append` operation. A single record in a stream can have multiple tag-value pairs. Similarly, a record can also have multiple corresponding to a single value. Tags and values are application-defined types that are required to

Function	Description
createStream(name, R/W, policy)	Create a data stream with specified policy properties (see Table 4.2)
openStream(name, R/W)	Open an existing data stream
deleteStream(name)	Delete an existing data stream
append([tag, value])	Append the list of values with corresponding tags. All values get the same timestamp
append([tag], value)	Append data labelled with potentially multiple tags
getLatest()	Retrieve latest tuple $< time, tag, value >$ inserted across *all* tags
get(tag)	Retrieve latest tuple $< time, tag, value >$ for the *specified* tag
getAll(tag)	Retrieve all time-sorted tuples $< time, tag, value >$ for the specified tag
getAll(tag, t_{start}, t_{end})	Range query: get all tuples for tag in the specified time range
getAll(tag, t_{start}, t_{end}, t_{skip})	Sampling range query
getKeys(tag_{start}, tag_{end})	Retrieve all tags in the specified *tag range*
sealStream()	Seal the current stream segment and create a new one for future appends
getAllSegmentIDs()	Retrieve the list of all segments in the stream
deleteSegment(segmentID)	Delete the specified segment in the current stream
grant(appId)	Grant appId read access
revoke(appId)	Revoke appId's read access

Table 4.1: Bolt stream APIs: Bolt offers two types of streams: (i) ValueStreams for small data values (e.g., temperature readings) and (ii) FileStreams for large values (e.g., images, videos).

implement specific contracts, which allows Bolt to hash, compare, and serialize them. Lastly, writers can `grant` and `revoke` the read access of other applications. Readers can filter and query data using tags and time, using the different `get` operations. Our prototype implementation supports querying for the latest record, the latest record for a tag, temporal range and sampling queries, and range queries on tags. Range queries return an iterator, which fetches data on demand.

Property	Description
Type	ValueStream or FileStream
Location	Local, remote, or remote replicated
Protection	Plain or encrypted
Sharing	Unlisted (private) or listed (shared)

Table 4.2: Properties specified by applications when performing a stream `create` or `open` operation.

4.4.2 Writing Stream Data

To write data to a stream, an owner first creates a new Bolt stream and appends data records to it. Figure 4.1 shows the data layout for a stream. A stream consists of two parts: a log of data records (called DataLog) and an index that maps a tag to a list of data item identifiers. Item identifiers are fixed-size entries and each list of item identifiers in the index is sorted by time, enabling efficient binary searches for range and sampling queries. The index resides in memory and is backed by a file; records in the DataLog are on disk and retrieved when referenced by the application. The DataLog is divided into fixed-size chunks of contiguous data records.

As large numbers of values are appended to a stream, the index can consume considerable amounts of memory. To reduce the memory footprint of the index, streams in Bolt can be archived. As described in Section 4.3, each stream is divided into segments, where each segment has its own DataLog and corresponding index. Hence, each stream is a time ordered list of segments. If the size of the stream's index in memory exceeds a configurable threshold ($index_{thresh}$), the latest segment is *sealed*, its index is flushed to disk, and a new segment with a memory resident index is created. Writes to the stream always go to the *latest* segment and all other segments of the stream are read-only entities. The index for the latest segment is memory-resident and is backed by a file

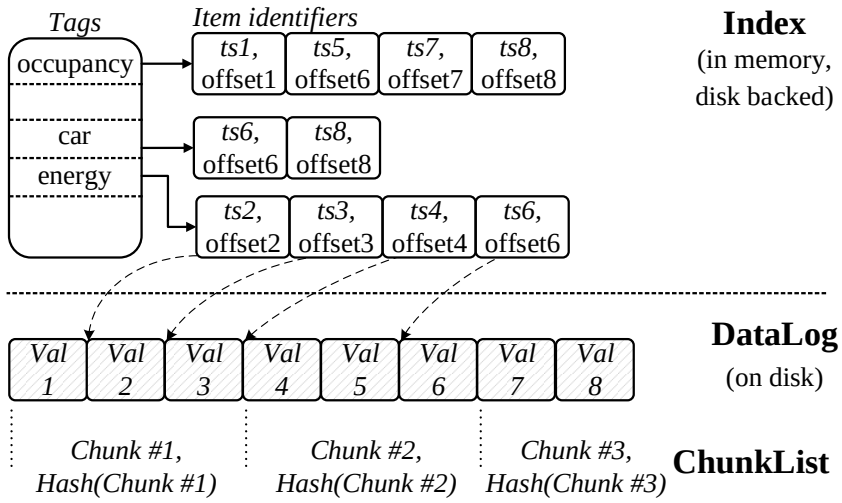


Figure 4.1: Data layout of a ValueStream. Data layout of a FileStream layout differs only in that the values in the DataLog are pointers to files that contain the data records.

(Figure 4.1). As shown in Figure 4.2, all other segments are sealed and store their indices on disk with a compact in-memory index header. The index header consists of the tags, the timestamp for the first and last identifier in their corresponding item identifier list, and the location of this list in the index file.

4.4.3 Uploading Stream Data

In Bolt, each reader or writer (identified by the <VHomeID, AppID> pair) is associated with a private-public key pair. Each stream in Bolt is encrypted with a secret key (denoted K_{con}) generated by the owner. When a stream is closed, Bolt flushes its index to disk, partitions the segment DataLog into chunks, compresses and encrypts these chunks, and generates the *ChunkList (CL)*. The per-segment ChunkList is an ordered list of all chunks in the segment's DataLog and their corresponding hashes. This operation is repeated for all *mutated segments*, that is, new segments generated since the last `close`, and the latest segment, since it may have been modified due to data appends. All other segments in a stream are sealed and are immutable.

Bolt then generates the stream's *integrity metadata (MD_{int})*. For a stream with n segments, its MD_{int} is computed as follows:

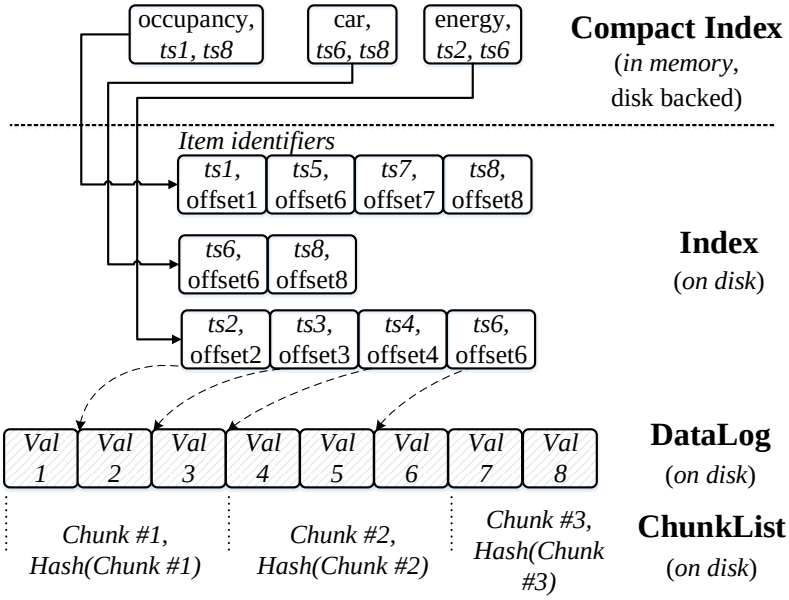


Figure 4.2: Data layout of a sealed segment in Bolt.

$$MD_{int} = \text{Sig}_{K_{priv}^{owner}} [\text{H}[\text{TTL}||\text{H}[\text{I}_i]||...||\text{H}[\text{I}_n]||\text{H}[\text{CL}_i]||...||\text{H}[\text{CL}_n]].$$

The `TTL` is a writer-specified consistency period which is used by readers to ensure that any data fetched from a storage server is no older the specified period, thus guaranteeing freshness of the data. For instance, when closing a given stream at 2013-09-23 10:10:00, a writer may specify TTL=(2013-09-23 10:10:00, 2013-09-23 10:20:00), implying that any stream data retrieved from the storage server is considered fresh until 2013-09-23 10:20:00, that is, the writer expects to write at least one value every 10 minutes. Other writers may similarly choose suitable TTL values while incorporating their expected write rate and maximum durations of any disconnections they expect.

As described in Table 4.3, MD_{int} is a signed hash of the TTL and the per-segment index and ChunkList hashes. For all mutated segments, Bolt uploads the chunks, the updated ChunkList, and the modified index to the storage server. Chunks are uploaded in parallel and applications can configure the maximum number of parallel uploads. The integrity metadata MD_{int} is also uploaded to (and resides on) a remote storage server.

Bolt uploads the stream metadata (described in the next section) to the metadata server. In our prototype implementation, the trusted metadata server (running on a

TTL: t^{start}_{fresh}, t^{end}_{fresh}
$H[x]$: Cryptographic hash of x
$Sig_K[x]$: Digital signature of x with the key K
K^{pub}_{owner}, K^{priv}_{owner}: A public-private key pair of the stream owner
CL_i: ChunkList of the i_{th} segment
I_i: Index of the i^{th} segment
$\|\|$: Concatenation

Table 4.3: Glossary.

personal VEE) stores the stream metadata so as to prevent unauthorized updates. We discuss potential solutions to relax this assumption in Section 4.7 .

4.4.4 Granting and Revoking Read Access

In addition to maintaining the mapping of readers and writers to their public-keys, the metadata server also maintains the following metadata for each stream: (i) a symmetric content key to encrypt and decrypt data (K_{con}), (ii) readers that have access to the data (including the owner) represented using their public keys, (iii) the storage location of the stream's integrity metadata MD_{int}, for example, Windows Azure, and (iv) per-segment location and key version. K_{con} values are stored in an encrypted form, one entry for each reader that has access to the stream, encrypted using the respective readers' public keys.

To grant read access to applications, the owner updates stream metadata with K_{con} encrypted with the reader's public key. To revoke read access the owner removes the appropriate reader from the accessor list, removes the encrypted content keys, and rolls forward the content key and key version for all valid principals by following the key regression protocol [84]. The key regression protocol allows readers with version V of the key to generate keys for all older versions 0 to V–1. As described in detail by Fu et al. [84], the protocol uses a chain of hash function computations to generate keys for older key versions from a given key. Therefore, each additional version only incurs the latency penalty of performing one fixed-size hash computation on the readers.

Upon a read revocation, Bolt seals the current segment of the stream and creates a new segment. All chunks in a segment are encrypted using the same version of the content key.

4.4.5 Reading Stream Data

Figure 4.3 illustrates the steps[3] for reading a stream with StreamID $S1$, by an application with AppID $A1$, running in VHome with VHomeID $V1$. Each stream is identified by the tuple <V1,A1,S1>. In step 1, Bolt opens a stream and fetches the stream metadata. It then uses the MD_{int}'s location information in the stream metadata to fetch MD_{int} from the remote untrusted storage server. After verifying MD_{int}'s integrity using the owner's public key, and data freshness using the TTL in MD_{int}, the reader fetches the index and ChunkList for every segment of the stream (step 2), and verifies their integrity using the respective hash values in MD_{int} (step 3).

Owners can append data records to a stream after they have verified the integrity of index data. In case of reads by non-owners, once the index and ChunkList integrity verifications for all segments are complete (step 3), Bolt uses the index to identify chunks that need to be fetched from one or more remote storage locations to satisfy an application's `get` requests.

Chunk level integrity is checked "lazily", that is, when the respective chunk gets downloaded for handling a query. Their integrity is verified by using the segment's ChunkList. Bolt decrypts and decompresses the verified chunk and caches these chunks in a local disk-based cache for subsequent reads.

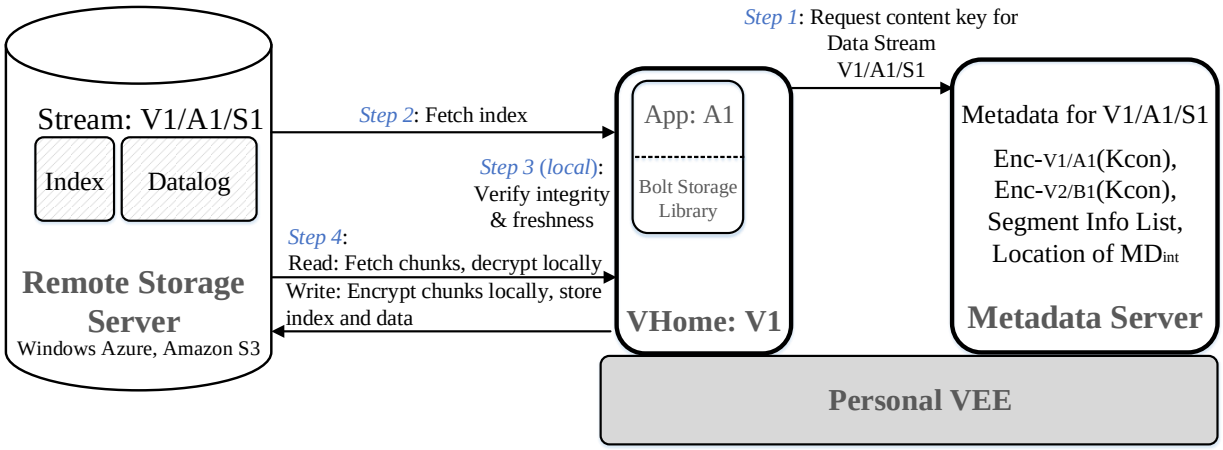


Figure 4.3: Steps followed during a read operation in Bolt.

[3]Stream owners also follow these steps when they re-open their streams.

4.5 Implementation

We have implemented Bolt using C# over the .NET Framework v4.5. Our implementation is integrated into the HomeOS [74] platform and can also be used as an independent library. In addition to the applications we evaluate in the next section, several other HomeOS applications have been ported by others to use Bolt. The client-side code is 6077 lines of code, and the metadata server is 473 lines. The current implementation is publicly available at `http://labofthings.codeplex.com`.

Our client library uses Protocol Buffers [12] for data serialization and can currently use Windows Azure and Amazon S3 for remote storage. It uses their respective libraries for reading and writing data remotely. On Windows Azure, each segment maps to an "Azure container", the index and DataLog each map to an "Azure blob", and individual chunks map to parts of the DataLog blob (called "blocks"). On Amazon S3, each segment maps to an "S3 bucket", the index maps to an "S3 object", and chunks of the DataLog map to individual objects. The communication between the clients and the metadata server uses the Windows Communication Foundation (WCF) framework.

4.6 Evaluation

We evaluate Bolt in two ways: (i) using microbenchmarks, which compare the performance of different Bolt stream configurations to the underlying operating system's performance (Section 4.6.1), and (ii) using applications, which demonstrate the feasibility and performance of Bolt in real-world use cases (Section 4.6.2). Table 4.4 summarizes the main results of the evaluation.

All Bolt microbenchmark experiments (in Section 4.6.1) are conducted on a virtual machine on Windows Azure, with a 4-core AMD Opteron Processor, 7 GB of RAM, one SATA hard disk with the NTFS filesystem, which is running the Windows Server 2008 R2 operating system. Similarly, all application workload experiments (in Section 4.6.2) are conducted on a physical machine with an AMD-FX-6100 6-core processor, 16 GB of RAM, one SATA hard disk with the NTFS filesystem, which is running the Windows 7 operating system. The metadata server also runs on the respective machines in the two cases.

Finding	Section
Bolt's encryption overhead is negligible, making its secure streams a viable default option.	Section 4.6.1
By using chunking, Bolt improves read throughput by up to 3 times for temporal range queries.	Section 4.6.1
Bolt segments are scalable: querying across 16 segments incurs only a 3.6% overhead over a single segment stream.	Section 4.6.1
Three applications (PreHeat, DNW, EDA) implemented using Bolt abstractions. Bolt performs 40 times faster than OpenTSDB for PreHeat while analysing 100 days of data. Bolt is 3 times more space efficient than OpenTSDB in storing 1000 days of PreHeat data.	Section 4.6.2

Table 4.4: Summary of Bolt's evaluation.

4.6.1 Microbenchmarks

Setup

In each microbenchmark experiment a dummy application issues 1,000 or 10,000 write or read requests for each of the three Bolt stream configurations: (i) using local disk as the storage location (denoted "Local"), (ii) using a remote storage provider (denoted "Remote") but with no data encryption, and (iii) using a remote storage provider with data encryption enabled (denoted "RemoteEnc"). We use Windows Azure as our example remote storage provider for all experiments.

The dummy application writes or reads data values of size 10 B, 1 KB, 10 KB or 100 KB. The chunk size for these experiments is fixed at 4 MB (unless otherwise specified). We also study the impact of varying chunk size later in this section. We measure Bolt's throughput in operations per second for different value sizes. For write operations, we also measure Bolt's storage overhead for different value sizes.

As a point of comparison for Bolt's local write performance, we conduct a separate set of experiments where data is written directly to the disk. We refer to this experiment as *DiskRaw*. For comparisons with Bolt's ValueStream, the DiskRaw write operations are to a single local file, and for comparisons with Bolt's FileStream, the DiskRaw write operations are to multiple files, one for each data value.

For data read operations, that is, `Get(tag)` and `GetAll(tag, time_start, time_end)` queries, we compare Bolt's read performance to data read directly from a single local file (in the case of ValueStream), data-values read from separate files (in the case of FileStream), and data read by downloading an Azure blob (in the case of remote ValueStream). All throughput values are reported as the mean of 20 repetitions with 95% confidence intervals.

Write Performance

We first present the microbenchmark results for ValueStreams (local and remote) followed by those for FileStreams (local and remote).

I. ValueStreams: Figure 4.4 compares the throughput of append operations (measured by the dummy application) for three different data-value sizes: 10 B, 1 KB, and 10 KB. We observe that the throughput of Bolt ValueStreams (marked Local, Remote, and RemoteEnc) is less than that of DiskRaw because the overhead of additional sub-tasks that Bolt has to perform: (i) index update/lookup, (ii) data serialization, (iii) writing index and DataLog to disk, (iv) uploading chunks and index (in the case of Remote and RemoteEnc), and (v) encrypting chunks (in the case of RemoteEnc).

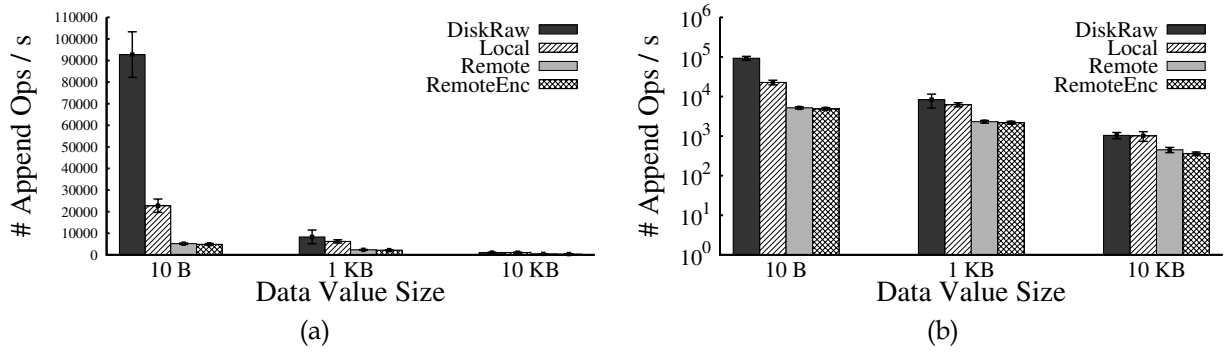


Figure 4.4: Write throughput for ValueStreams shown using a linear scale in (a) and a logarithmic scale in (b).

Table 4.5 shows the percentage of time taken to perform these sub-tasks in a sample experiment of 10,000 append operations of 10 byte values. We find that the time taken to perform the additional sub-tasks (i), (ii), and (iii) described above, in the case of Remote and RemoteEnc is similar to that in case of local ValueStreams. However, in the case of

Remote and RemoteEnc, a high percentage of the total time is spent uploading the DataLog and index to the cloud. For instance, approximately 72% of the total time is taken to perform uploading of the DataLog while uploading the index consumes approximately 3% of the total time. Uploading data chunks involves six main components: (i) reading chunks from the DataLog and computing the hash of their contents, (ii) checking the chunk's existence on the remote storage server (as an Azure Blob) and creating one if not present, (iii) compressing and encrypting chunks (if enabled), (iv) uploading individual chunks to blocks in an Azure Blob, (v) committing the new block list reflecting the new changes to Azure's storage servers, and (vi) uploading the new chunk list containing chunk IDs and their hashes. Also note that, in the case of RemoteEnc, the time taken to encrypt the data chunks is less than 1% of the total time.

Component	Local	Remote	RemoteEnc
Lookup, update Index	6.8 %	2.7%	0.9%
Data serialization	15.2 %	2.6%	1.9%
Flush index	42.5 %	11.3%	11.5%
Flush DataLog	35.4 %	8.3%	11.8%
Uploading chunks	-	72.0%	70.2%
Encrypting chunks	-	-	0.7%
Uploading index	-	3.1%	3.0%

Table 4.5: Percentage of total experiment time spent in various tasks in a sample experiment of 10,000 append operations (of 10 byte values) to a ValueStream.

In Figure 4.4, we also observe that the measured throughput for both Bolt and DiskRaw decreases with increasing data value sizes. This is because the increasing data value size increases the amount of data written to the local disk per second which saturates the local disk write throughput and decreases the throughput of append operations.

Table 4.6 shows the storage overhead of ValueStreams as compared to DiskRaw for data value sizes of 10 B, 1 KB, and 10 KB. We define *storage overhead* as the amount of additional disk space used by a ValueStream as compared to DiskRaw, expressed as a percentage. Note that, in the case of DiskRaw, tag-value pairs and timestamps are appended to a file on local disk. Bolt incurs storage overhead due to its storage of offsets in the index, and additional data for serializing and de-serializing the index. As described in Section 4.4.2 (Figure 4.1), Bolt stores each unique tag only once in the index, thus benefiting streams with relatively large tags. Bolt's storage overhead decreases with increasing value sizes but remains constant with increasing number of data records for a given value size.

Value size	Percentage overhead
10 B	30.6 ± 0.0
1 KB	0.86 ± 0.0
10 KB	0.09 ± 0.0

Table 4.6: Storage overhead of local ValueStreams as compared to DiskRaw.

II. FileStreams: Figure 4.5 compares the write throughput for 1000 appends for three different data-value sizes (1 KB, 10 KB, and 100 KB). The time taken for an append operation in a FileStream is dominated by two main sub-tasks: (i) writing each data value to a separate file on disk, and (ii) uploading each file to a separate Azure blob (in the case of remote FileStreams).

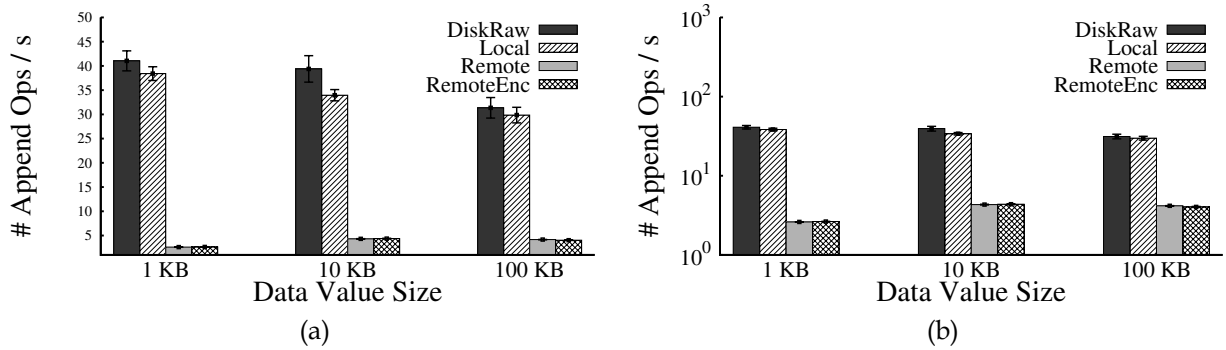


Figure 4.5: Write throughput for FileStreams shown using a linear scale in (a) and a logarithmic scale in (b).

We observe that in the case of a local FileStream the write throughput is comparable to that of DiskRaw. In the case of remote streams, time taken to create a new Azure blob for every data value dominates the time taken for a write operation and in our experiments accounts for 80% of the total time for an append operation.

We also observe that, similar to ValueStreams, encryption overhead in the case of remote-encrypted FileStreams is approximately 1% of total time. Lastly, the storage overhead in the case of FileStreams is approximately the same as that of ValueStreams with a value size of 10 bytes. This is because FileStreams store pointers (each 8 bytes) to data files instead of offsets into the DataLog (as in ValueStreams).

Read Performance

We first present the microbenchmark results for ValueStreams (local and remote) followed by those for FileStreams (local and remote).

I. ValueStream: Figure 4.6 compares the read throughput for three different data-value sizes (10 B, 1 KB, and 10 KB) for the different ValueStream configurations. The dummy application issues $10,000$ `Get(tag)` requests with a uniform randomly selected tag for each request. In the case of DiskRaw, the values are read from random parts of the DataLog. Despite random disk read operations issued both in DiskRaw and ValueStreams, we believe that the file system cache affects throughput measurements in both cases.

We observe that local ValueStreams have a lower read throughput than DiskRaw. This is because each read operation in a ValueStream incurs a latency overhead due to index lookup and data deserialization. For instance, for 1 KB sized data values, we find that local ValueStreams use 5% of the total time for index lookup, 60% of the total time for reading records from the DataLog (which matches DiskRaw), and 30% of the total time for deserializing data. In case of remote ValueStreams, remote reads' latency (and hence throughput) are dominated by the time taken for downloading DataLog chunks from Azure and storing them in the local chunk cache (approximately 90% of the total time for `Get(tag)`).

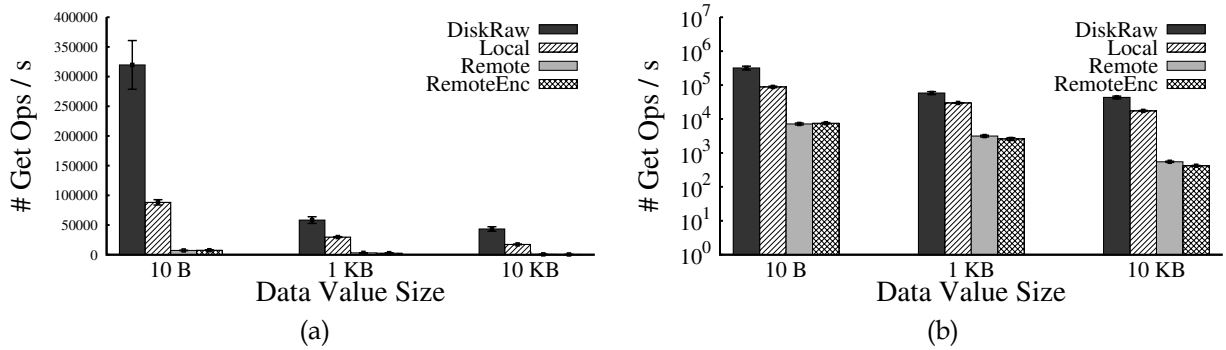


Figure 4.6: Read throughput with randomly selected tags for ValueStreams shown using a linear scale in (a) and a logarithmic scale in (b).

Effect of Chunk Size on Temporal Range Queries: To demonstrate the effect of chunking on query latency and throughput of remote ValueStreams we present two example range queries. The first query retrieves a window of 10 records whereas the second

query has a window of 100 records. The start and end times of the windows are picked randomly from the time range of data stored. We choose an example stream with 10,000 data values of 10 KB each.

Figure 4.7 shows the read throughput with increasing chunk size. We observe that chunking reduces latency of reads and hence improves read throughput because it batches transfers and prefetches data for range queries with locality of reference. We observe higher read throughput at relatively large chunk sizes because of the reduction in the number of chunk downloads because chunks are cached locally. For a given chunk size the 100-record query has slightly higher throughput than the 10-record query. This is because the queries' start times are uniform randomly selected and hence wider queries benefit more from the local chunk cache when answering queries, thus causing relatively few chunk downloads. On the other hand, narrow queries are comparatively dispersed across the DataLog, hence causing relatively more chunk downloads, which increases query latency and decreases throughput.

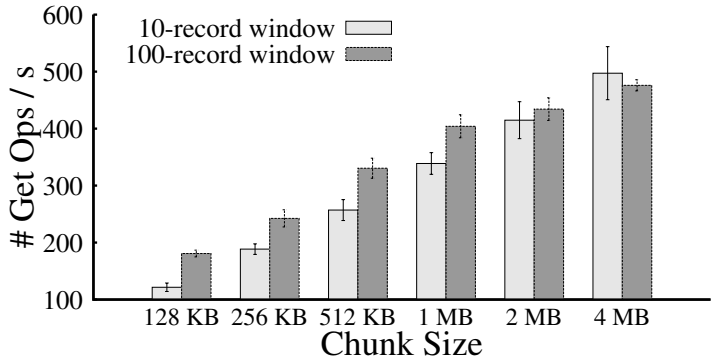


Figure 4.7: Read throughput for remote ValueStreams with varying chunk size for two sample `Get(tag,`t_{start}`,`t_{end}`)` queries with locality of reference.

Effect of increasing number of ValueStream Segments: Figure 4.8 shows the effect of increasing the number of segments for a local ValueStream on the time taken to open the stream (one time cost), index lookup, and reading data records. Each segment has $10,000$ keys, and $10,000$ `Get(tag)` requests are issued for uniform randomly selected keys. The time taken for opening a stream is dominated by the time to build the segment index in memory and it increases significantly with the increasing number of segments. The time taken for index lookup and reading the data records also increases with the number of segments because all segments except the last have a compact memory-resident index header, and as explained in Section 4.4.2, require additional disk reads for index lookups.

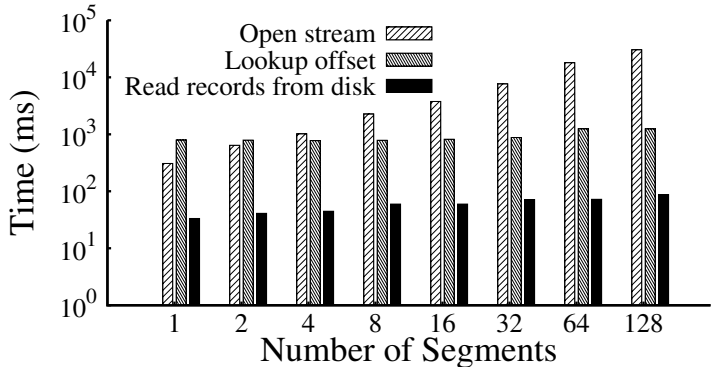


Figure 4.8: Open, index look-up, and DataLog record retrieval latencies (on the logarithmic scale) with increasing number of segments of a local ValueStream, measured by issuing $10,000$ `Get(tag)` requests for randomly selected keys.

II. FileStreams: Figure 4.9 compares the read throughput for three different data-value sizes (10 B, 1 KB, and 10 KB) for the different FileStream configurations. Recall that, in the case of FileStreams, individual data values are stored in separate files on disk and are uploaded to separate blobs on remote storage.

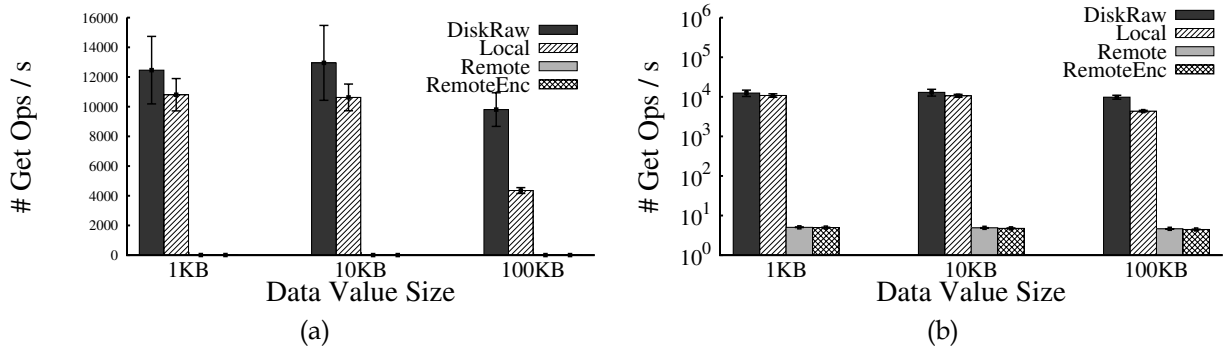


Figure 4.9: Read throughput with randomly selected tags for FileStreams shown using a linear scale in (a) and a logarithmic scale in (b).

As compared to DiskRaw, FileStreams incur the additional latency overhead of index lookup, downloading individual blobs from remote storage, and reading each data record from its individual blob. These additional operations significantly impact the

read throughput of FileStreams. In the case of remote FileStreams, a large fraction of the read time (approximately 99%), is spent downloading the individual blobs from remote storage. It may be possible to improve the read throughput of FileStreams by storing multiple data values (which are stored in separate files on local disk) using a single blob on the remote storage server (as in the case of ValueStreams). Lastly, we also find that, similar to ValueStreams, in the case of remote-encrypted FileStreams, the time taken to decrypt data is less than 1% of the total get-operation time.

4.6.2 Applications

We instrument the three real-world applications described in Section 4.2.1 using Bolt, and measure their query latencies. For comparison we also evaluate the query latencies of these applications using OpenTSDB [27], which is a popular time-series data storage and analytics system. It is written in Java and uses HBase to store data. Unlike Bolt's library that is loaded into the client application program, OpenTSDB only allows querying only over HTTP endpoints. Moreover, unlike Bolt, OpenTSDB provides neither the data security guarantees nor the flexibility of policy-driven storage across providers.

We now present detailed results for each application. All latency values are reported as the mean of 20 repetitions with 95% confidence intervals, using Windows Azure as the remote storage provider.

PreHeat

Recall that PreHeat [161] is a system that enables efficient home heating by recording and predicting occupancy information. We describe PreHeat's algorithm in Section 4.2.1. To implement PreHeat's data storage and retrieval using Bolt we identify each slot by its starting timestamp. A single *local* unencrypted ValueStream thus stores the *(timestamp, tag, value)* tuples where the *tag* is a string (e.g., "occupancy"). We built two different PreHeat implementations that either seek to minimize the amount of data stored on disk or data-retrieval time and hence store different data values:

Naïve + ValueStream: In this implementation, the value stored for each 15-minute slot is simply the slot's measured occupancy (0 or 1). Therefore, for computing predicted occupancy for the nth slot on day d, POVs are obtained by issuing d temporal range queries (using `getAll(`k`,` t_s`,` t_e`)`).

Smart + ValueStream: In this implementation, the value stored for each 15-minute slot is its POV concatenated with the measured occupancy value in that slot. As described in Section 4.2.1, computing the predicted occupancy for the nth slot on day d requires obtaining POV_d^n, POV_{d-1}^n, ..., POV_1^n. Thus, under this scheme, these required POVs can be obtained by issuing: (i) one `get(tag)` query for the $(n-1)$th slot on day d to obtain POV_d^n, and (ii) $(d-1)$ temporal range queries that obtain the values stored for the nth slot on days $(d-1)$, $(d-2)$, ..., 1, thus obtaining the values of POV_{d-1}^n, POV_{d-2}^n, ..., POV_1^n respectively. Therefore, in this implementation, the range queries present in the naïve implementation are replaced with simple get queries that obtain values stored for particular slots. However, the storage overhead incurred with this approach is larger than the naïve approach but the retrieval latency is reduced due to the reduced number of disk reads.

Naïve + OpenTSDB: We implement the naïve PreHeat approach to store and retrieve data locally from OpenTSDB. It groups occupancy values spanning an hour into one row of HBase (OpenTSDB's underlying datastore). That is, four PreHeat slots are grouped into a single HBase row. OpenTSDB's usability is limited by values being restricted to real numbers. Bolt allows byte arrays of arbitrary length to be stored as values. Consequently, an analogous implementation of the smart PreHeat approach is *not feasible* using OpenTSDB.

Of all the 96 slots in a day, the 96th (last slot) has the maximum retrieval, computation, and append time since POV_{96}^d is longest $POV_i^d, \forall i \in [1, 96]$. Therefore, to compare the Naïve+ValueStream, Smart+ValueStream, and Naïve+OpenTSDB approaches, we use the retrieval times for the 96th slot of each day. Figure 4.10 shows the time taken to retrieve data for the 96th slot for Naïve+ValueStream, Smart+ValueStream, and Naïve+OpenTSDB.

We observe that as the number of days increases the retrieval latency for both the naïve and smart approaches grows due to the increasing number of range and get queries. However, the smart implementation incurs lower latency than the naïve implementation because it issues fewer random disk reads.

When comparing the Naïve+ValueStream and Naïve+OpenTSDB implementations for retrieving 100 days of data, we observe that Bolt performs approximately 40 times faster. As expected, the time taken to perform the occupancy prediction computation after retrieving the required data is unchanged across the three implementations. Table 4.7 shows the storage space required by the different implementations for 1000 days of operation. We observe that the smart approach uses approximately 8 times the storage space compared to naïve because it also stores the slots' POVs in addition to their

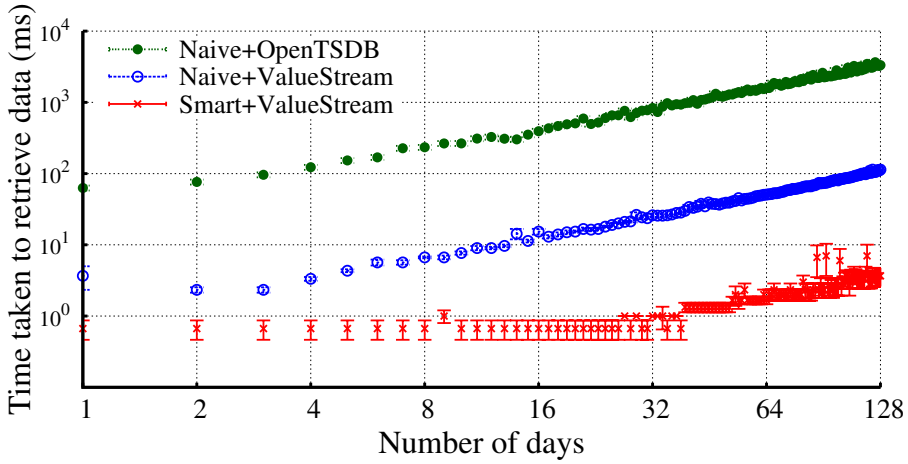


Figure 4.10: Time to retrieve past occupancy data with increasing duration of a PreHeat deployment.

occupancy values. Using Bolt's compressed streams, we observe that the naïve scheme reduces the storage overhead by a factor of 1.6 and the smart scheme reduces the storage overhead by a factor of 8, when compared to their uncompressed stream counterparts. OpenTSDB requires a 3 times larger disk footprint than its corresponding implementation using Bolt with compression turned off. We believe that row key duplication in HBase is the potential source of this storage inefficiency.

Configuration	Naive	Smart
ValueStream	2.38 MB	19.10 MB
ValueStream using GZip	1.51 MB	3.71 MB
ValueStream using BZip2	1.48 MB	2.37 MB
OpenTSDB	8.22 MB	Not feasible

Table 4.7: Storage space required for a 1000-day deployment of PreHeat.

Digital Neighborhood Watch (DNW)

As described in Section 4.2.1, DNW helps neighbors detect suspicious activities (e.g., an unknown car cruising the neighborhood) by sharing security camera images [56]. We implement DNW's data storage, retrieval and sharing mechanisms using Bolt and

OpenTSDB. Due to Bolt's (timestamp, tag, value) abstraction, objects can be stored (and retrieved) in a single remote ValueStream (per home), with each object attribute in a separate tag, such as, type, ID, and feature-vector, all bearing the same timestamp. Queries proceed by first performing a `getAll(feature-vector, ts, te)` where the time window (w)=[t_s,t_e], and then finding a match amongst the retrieved objects. In contrast, with OpenTSDB, each home's objects need to be recorded using specialized OpenTSDB tags (called "OpenTSDB metrics"), which also store the object attributes. As described in Section 4.2.1, we simulate DNW as a cloud-based VHome application. Therefore, in our DNW experiments, we run OpenTSDB remotely on a virtual machine on Windows Azure and the DNW client applications run on a physical machine (described above). Similarly, in case of Bolt, the DNW client applications run on the physical machine and use remote ValueStreams (with Windows Azure as the storage provider).

To evaluate DNW, we instrument a scenario where each home (i.e., a client application) records an object every minute. After 1000 minutes, one randomly selected client application queries all other homes for a matching object within a recent time window w. We measure the total time for retrieving all object summaries from ten homes for window sizes of 1 hour and 10 hours.

Figure 4.11(left) shows the retrieval time for DNW with increasing chunk sizes (for Bolt) for time windows of 1 hour and 10 hours. We observe that, in the case of Bolt, larger chunk sizes reduce retrieval time by batching transfers for the windows size of 10 hours. For queries that span multiple chunks, Bolt downloads data chunks on demand since range queries return an iterator to the application (Section 4.4.1). When applications use this iterator to access a data record, Bolt checks if the corresponding chunk for the data record is present in the cache and if not it is downloaded. Hence, queries that span multiple chunks require multiple round trups as chunks are downloaded on-demand, thus increasing the overall retrieval time. Example queries include one where DNW runs for 10 hours with a 100 KB chunk size. This can be improved by pre-fetching chunks in parallel, in the background, without blocking the application's range query. In the case of large chunk sizes, fewer chunks need to be downloaded (shown in Figure 4.11(center)) resulting in fewer round trips which reduces the overall retrieval time for the 10-hour query. Note that OpenTSDB does not implement chunking. Hence OpenTSDB retrieval times in Figure 4.11(left) are independent of chunk size.

In the case of Bolt, we also observe that when chunk size becomes large enough, a large amount of additional data fetched in the chunk does not match the query. Consequently, the decrease in retrieval time that is obtained by batched transfers becomes relatively low at large chunk sizes. Moreover, since chunk boundaries do not necessarily line up with the time window specified in queries, data records that do not match the

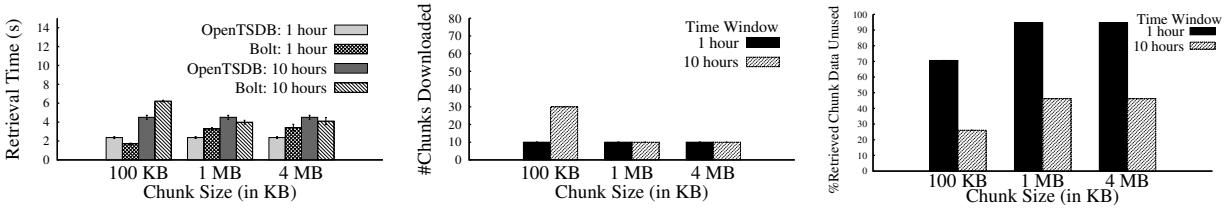


Figure 4.11: Retrieving object summaries in DNW from 10 homes for time windows of 1 hour and 10 hours using Bolt and OpenTSDB. Retrieval times for OpenTSDB are independent of chunk size.

Storage system	DNW	EDA
Bolt	4.64 MB	37.89 MB
OpenTSDB	14.42 MB	212.41 MB

Table 4.8: Storage space used for 10 homes in DNW for 1000 minutes, and 100 homes in EDA for 545 days.

query may be fetched even for small chunk sizes. Figure 4.11(right) shows that as chunk size increases, the percentage of data records in chunks downloaded that do not match the query's time window w also increases. Bolt allows applications to choose chunk sizes to match their workloads by trading this overhead for retrieval time. Lastly, Table 4.8 compares the storage footprint of DNW using Bolt with OpenTSDB. As is the case with PreHeat, we observe that Bolt incurs a 3 times smaller disk footprint than OpenTSDB.

Energy Data Analytics (EDA)

As explained in Section 4.2.1, we study a scenario where a cloud-based application presents consumers with an analysis of their energy consumption by comparing their consumption with other homes in the neighborhood, or city, within a given time window such as one month.

In the implementation using Bolt, we use a single remote ValueStream per home which stores the smart meter data. For each hour, the energy consumption value is appended to the ValueStream with the mean ambient temperature value of the hour (rounded to the nearest whole number) as the tag. This allows us to easily retrieve all energy consumption values for a given temperature using a single `get(tag,`t_{start}`,`t_{end}`)` query. In the OpenTSDB-based implementation, we need to create one tag (called "OpenTSDB

metric") for each temperature value, for each home; that is, a record *home-n-T* stores values recorded at temperature T for home n. For each home we retrieve data in the time interval $[t_s, t_e]$ for each temperature T between -30° C and 40° C (the range of observed ambient temperatures in the dataset). The retrieved data is used by the application to compute the median, 10th, and 90th percentile energy consumption values for a home which is then compared with all the other homes being considered. Data for multiple homes is retrieved sequentially.

Figure 4.12 shows the time taken to retrieve data as we increase the number of homes, for two time windows of 1 month, and 1 year. We use an actual home energy consumption dataset from a utility company in Ontario, Canada. Retrieval time for Bolt and OpenTSDB increases proportionally at approximately 1.4 seconds per home and 11.4 seconds per home respectively, for a one month window. In the case of a one year window the retrieval time for Bolt and OpenTSDB increases at approximately 2.5 seconds per home and 12 seconds per home respectively. This is because of more data being fetched in case of a 1 year window query than that in a 1 month window query. Bolt's retrieval time outperforms OpenTSDB by an order of magnitude, primarily due to prefetching data in chunks, which is explained as follows.

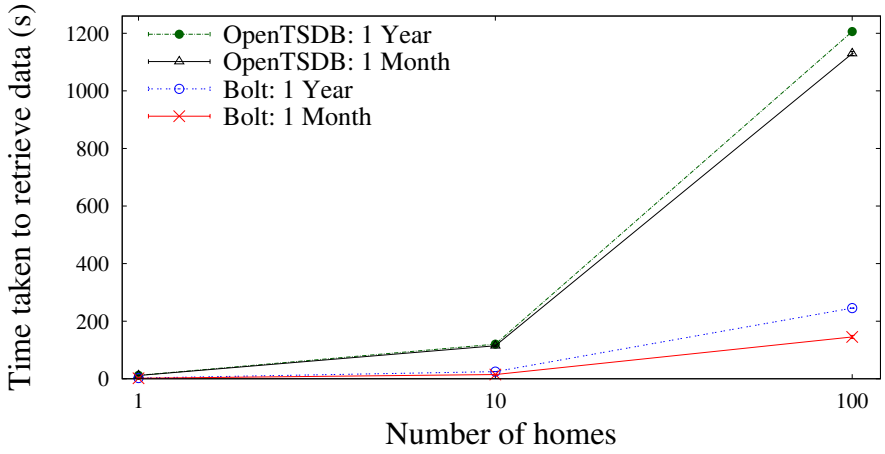


Figure 4.12: Time taken to retrieve smart meter data for multiple homes for time windows of 1 month and 1 year.

As explained in Section 4.4, Bolt stores data records in a time-sorted DataLog and the index maps each tag (in this case, each temperature value) to offsets into the DataLog. The DataLog is then partitioned into chunks and all data transfers to and from remote storage servers happen on a chunk basis. Therefore, the set of data chunks downloaded

for answering a range query for a given tag such as 10°C in the case of EDA, will often overlap with that required for handling a range query for an adjacent temperature value tag such as 11°C. This is because hourly ambient temperature values typically do not vary significantly over time, for example, never more than ±2°C in our dataset. Therefore, Bolt is able to effectively leverage its local chunk cache when answering successive range queries for a set of tags such as {-30°C, -31°C, …, 40°C}, and thus incurs a lower retrieval time than OpenTSDB. However, it may be possible to improve the retrieval time for OpenTSDB by extending it with a caching mechanism on the client-side.

Finally, as shown in Table 4.8, we observe that OpenTSDB incurs a 5 times larger disk footprint than its corresponding implementation using Bolt.

4.7 Discussion

We discuss two ways to extend Bolt to reduce its reliance on the trusted metadata server assumptions and improving Bolt's sharing flexibility.

Relaxing the assumption of the trusted key server

Bolt's current design includes a trusted metadata server to: (i) prevent unauthorized changes to the mapping of readers and writers to their public-keys, and (ii) prevent unauthorized updates (rollback) of the key version for each segment of a stream. Unauthorized changes to the public-key mappings may trick the owner of a stream to grant access to a malicious reader. Similarly, unauthorized updates of the key versions may cause the owner to use an invalid content key to encrypt data, potentially exposing the newly written content to readers whose access has been already revoked.

Bolt also relies on the metadata server to distribute the keys and the stream metadata. However, a few extensions can be built to minimize this trust dependency. One approach is to replicate the information stored at the metadata server at $2f + 1$ servers and use a majority to tolerate up to f malicious servers. An alternate solution would be to employ a Byzantine quorum system, to tolerate up to a third of the servers being compromised at any given time.

Supporting finer-grained sharing

In our current prototype, readers are granted access to the entire stream. Once their read access has been revoked they cannot access any new segments of the stream created since the revocation, although they could still access all the previous segments. Bolt can potentially support finer-grained read access by creating a different key for each segment. This approach trades off metadata storage space for segment-level sharing.

4.8 Chapter Summary

Our prototype storage system in Chapter 3 suffers from two main shortcomings. First, many applications require richer data manipulation capabilities than those supported by our prototype data storage system. Second, our prototype storage system only uses local VEE storage, while users and applications may also want to use commodity cloud storage services, for example, to lower storage costs or obtain better reliability. Therefore, our goal is to simplify the management of data from in-home sensors and build a suitable data management system that replaces our prototype storage system. To do so, we survey a range of home applications and formulate the requirements they place on the storage system. We find that existing systems do not meet all of these requirements. Therefore, we design Bolt, a data management system for such applications. It provides applications with a timestamp-tag-value stream abstraction for storing and retrieving data, allows application instances to easily share data, allows applications to control storage of data across commodity storage providers such as Windows Azure and Amazon S3. At the same time, Bolt provides data confidentiality guarantees and data integrity by providing evidence of data tampering and freshness guarantees. We evaluate our current Bolt implementation using microbenchmarks and three example applications. We observe that in our sample applications Bolt has up to 40 times lower data retrieval latency and incurs up to 3-5 times lower storage space when compared to OpenTSDB [27].

Chapter 5

Provisioning Large Numbers of Personal VEEs

5.1 Introduction

Our personal VEE architecture (introduced in Chapter 1) requires provisioning one VEE for each user, thus requiring large numbers of VEEs to be provisioned. For instance, if every home with a smart meter was to be provided with an instantiation of this architecture, 36 million VEEs in the US [42] (4 million in Ontario, Canada [41]) would have to be provisioned. Similarly, providing every active Facebook user with data-driven applications using this architecture will require 1.4 billion [11] VEEs.

In our prototype instantiations in Chapters 3 and 4, we used virtual machines (VMs) from cloud providers, such as Windows Azure to realize personal VEEs. Most cloud providers use commodity virtualization solutions (described in Section 2.3.1; Table 2.2) to provision such VMs. Moreover, they employ VM consolidation methods [71,103,123, 190] that allow them to pack multiple VM instances running on underutilized physical machines into fewer machines, enabling some machines to be turned off to obtain energy savings.

Typical VM consolidation methods use VM migration and simply re-pack VMs into fewer physical machines based on the pre-configured resource capacities of the VMs. Therefore, the *VM density* (average number of VMs/machine) such methods yield is bounded by the maximum number of VMs that can be co-hosted on a single machine.

For instance, Citrix XenServer claims to provision 100-200 VMs/machine[1]. Provisioning large numbers of VMs (e.g., 1 billion) using this approach will require un-affordably large amounts of hardware resources (e.g., 5 million machines for 1 billion VMs).

However, many VM workloads exhibit frequent, often long, and uncorrelated idle periods. Personal VEE workloads are an example of this type of workload. This is because applications hosted on personal VEEs typically cater to a small set of users, VEE-hosted jobs are either user-initiated, such as data analytics and sharing applications [77,137], or are periodic with relatively large periods (of the order of seconds), such as periodic data batch uploads and data pre-processing. Other such example workloads include some web-hosting [118] and cyber-foraging workloads [158].

When multiple VMs with such workloads are co-hosted on a machine, decreasing the resource footprint of idle VMs allows for a much denser VM packing, thus reducing the required hardware resources and hosting costs for both users and cloud-providers. For such workloads, it is possible to reclaim resources from idle VMs by transitioning them to *inactive state(s)* and activating them on the arrival of client requests. This leads to increased VM density at the cost of an increase in client request latency (called the *miss penalty*). Such a consolidation effort is compatible with existing migration-based consolidation methods since it incurs little network overhead. Moreover, this type of consolidation is able to leverage transient idle periods to reclaim resources whereas conventional VM migration-based consolidation methods rely solely on long idle periods.

As described in Section 2.3.3, many inactive states have been proposed or are supported in existing virtualization solutions (shown in Table 2.3). Recent work [94,118] has explored the use of one inactive state for managing idle VMs. In doing so, VM density is limited by the maximum number of VMs (per machine) that can be simultaneously hosted in that inactive state (e.g., substrate [179], fast-resume [187, 188]) on a machine. If more than one such inactive states were used to host the idle VMs, VM density can be further increased. Unfortunately, due to differences in their design and resource requirements, different inactive states have varying VM activation and deactivation times and VM capacities. Consequently, miss penalties for idle VMs in different inactive states vary significantly. This leads to the following questions.

1. How should idle VMs be transitioned across different inactive states? In other words, when a VM becomes idle which inactive state should it be transitioned to?

2. Subsequently, when should an idle VM be transitioned to other state(s) in anticipation of client requests?

[1] Using a particular hardware configuration, and subject to VM sizes and workloads.

It is clear that a policy that governs the transitions of idle VMs across inactive states is needed. Ideally, such a policy should minimize the miss penalties and maximize VM density. We refer to these policies as *idle VM management policies*. Existing mechanisms (such as those discussed in Section 2.3.3) can then be used to implement such policies, by determining VM idleness and activeness and dynamically transitioning VMs between the active and various inactive states.

In this chapter we study the effect of different idle VM management policies on VM density and miss penalties. To do so, we formally model the problem of multiplexing idle VMs across multiple inactive states. We divide the policy space into two parts, (i) *demand-based* (or reactive) policies, and (ii) *proactive* policies. Using our model formulation, we provide a lower-bound on the miss penalty incurred by demand-based policies. Then, by finding similarities between this problem and the problems of page replacement and multi-level cache management, we propose *SlidingWindow*, a proactive policy which leverages inter-arrival time prediction to reduce miss penalties. We measure the model parameters for Linux Containers (LXC) [22], a widely used OS-level virtualization solution. We then use the measured parameters for LXC to evaluate different reactive and proactive policies using some example personal VEE workloads.

Our key contributions in this chapter are:

- A formal model for idle VM management policies and a lower bound on the miss penalty of reactive policies.
- A measurement of model parameters using microbenchmarks with LXC [22] as our example virtualization solution.
- A study of a few representative VM management policies quantifying their miss penalties using a simulation-based evaluation and providing insight into their behaviour.

This work has been published in the proceedings of ACM VEE 2015 [167].

The remainder of the chapter is structured as follows. We formally model the problem in Section 5.2 and derive a lower bound on miss penalties of reactive policies. We present the measurement of model parameters for LXC [22] using microbenchmarks in Section 5.3. We describe our setup for comparing different policies and our example workloads in Section 5.4. We evaluate different policies using simulations in Section 5.5 and present a discussion of our work in Section 5.7. We conclude with a summary of the chapter in Section 5.7.

5.2 Problem and Model Formulation

Typical cloud environments today exhibit a great deal of VM heterogeneity. That is, VMs have varying resource capacities, serve disparate workloads, and provide varying service level agreements (SLAs) to their tenants. In this work, as a starting point, we study scenarios where VMs have relatively homogeneous resource capacities, workloads, and SLAs, such as VMs used as personal VEEs. We defer the study of heterogeneous scenarios to future work.

Consider a VM that is provisioned on a given physical machine and is initially in the booted state. Once such a VM becomes idle it can either be left in the booted state, or it can be transitioned (either immediately or at a suitable later time) to one of the inactive states, depending on the VM management policy in place. Since different inactive states, due to differences in their design and resource intensity, have different transition-to-booted times, the policy's actions can significantly affect miss penalties for subsequent requests for this VM. Similarly, because the maximum number of VMs possible in each state is limited (typically by a specific system resource), the policy's actions may also affect miss penalties for other VMs depending on which state it chooses to place them in. Due to such implications on miss penalties, it is important to choose a VM management policy that minimizes miss penalties across all VMs while maximizing VM density.

To better understand and reason about different possible policies, we formulate a simple mathematical model of the problem. Such a formulation provides a sound theoretical foundation for the problem, and as we show, can be used to provide a lower bound on the miss penalty incurred by any demand-based policy.

Let $S_1, S_2 \ldots S_n$ (as shown in Figure 5.1) be the n inactive states. In addition let S_0 be the booted state, and S_{n+1} be the shutdown state. Similarly, let $V_1, V_2 \ldots V_v$ be v VMs provisioned on a machine. Let the maximum number of VMs feasible in state S_i be B_i. Let matrix $T_{(n+2)\times(n+2)}$ be the time to transition VMs between two states, that is, $t_{i,j}$ is the time to transition from state S_i to state S_j where $i, j \in \{0, 1, \ldots n, n+1\}$. We realize that in practice, the transition times may be stochastic variables (associated with some distribution), for example, depending on the number of VMs in different states at any point. Our model can be viewed as a mean value analysis and can be extended to conduct a stochastic value analysis. We view B_i as a soft bound, that is, if the number of VMs in S_i exceeds B_i, transition times ($\forall j$, $t_{i,j}$ and $t_{j,i}$) may degrade. To simplify the notation in a later proof, let t'_i be the time to transition from S_i to S_0 (booted), that is, $t'_i = t_{i,0}$ for $i \in \{0, 1, \ldots n, n+1\}$.

Let VM requests received over any fixed period of time be denoted as a string ω

comprised of tuples:

$$\omega = r_1, r_2 \ldots r_{|\omega|}$$

where $r_i = (V_j, t_i, d_i)$, t_i denotes the time at which the request r_i arrives for the *target VM* V_j, and d_i denotes the duration for which VM V_j remains active to service r_i. We refer to ω as the *request string*. For a given ω, let $P_\pi(\omega)$ denote the total miss penalty incurred by VM management policy π. Thus for an optimal VM management policy θ, $\forall\omega\forall\pi, P_\theta(\omega) \leq P_\pi(\omega)$.

Policies can be divided into two classes: (i) **reactive (or demand-based)** policies, and (ii) **proactive** policies. We now address each class separately and describe how VM management policies relate to other types of resource management policies.

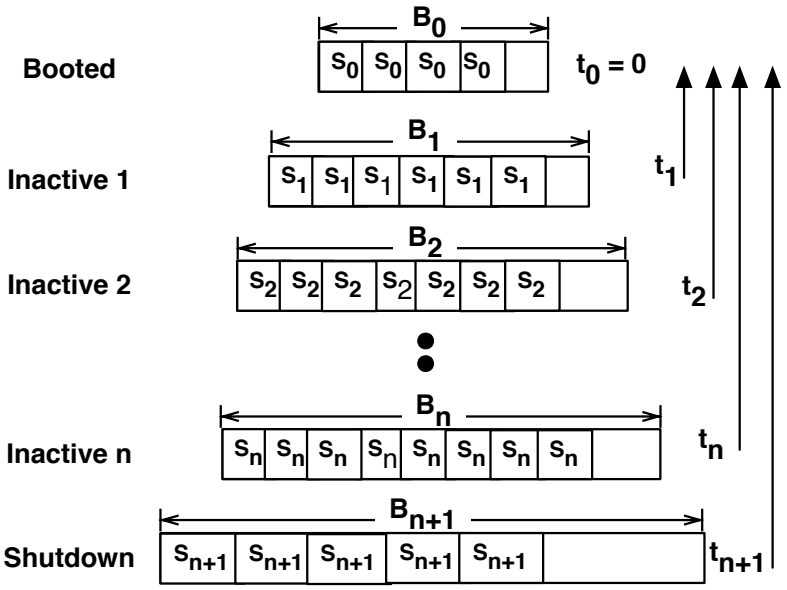


Figure 5.1: Hierarchy of booted, shutdown, and n inactive states.

5.2.1 Reactive Policies

Reactive policies configure, transition, or provision a system resource, such as a memory page, only when a demand for that resource is received, and are also referred to as *demand-based* policies. Examples include the widely used demand-based page replacement policies such as LRU, FIFO, Clock, and others [70]. Another prominent example is multi-level cache management policies such as DEMOTE [183]. Belady's OPT

algorithm [52], which simply evicts the page referenced furthest in the future, is provably the optimal demand-based policy for single-level caches [47,88] (e.g., page replacement). However the optimal demand-based policy for multi-level caches remains unknown [88]. We believe that finding the optimal demand-based policy for exclusive multi-level caches [88] can help uncover the optimal demand-based policy for idle VM management[2].

Note however that idle VM management differs from page replacement and cache management in a few ways. For example, page replacement policies are able to leverage locality-of-reference; in contrast, each VM has a separate tenant with little locality-of-reference across VMs. Each VM request must guarantee that the target VM remains in the booted state at least for the duration of the request; page references require no such guarantees.

Nevertheless, we build upon existing results from page replacement and caching policies and formulate a lower bound on the miss penalty $P_\pi(\omega)$ incurred by any demand-based VM management policy π. Policies can be either *online*, which only use information about the past, or *offline*, which also use information about the future. Our lower bound assumes knowledge of future arrivals and cannot be implemented in an online fashion. It serves as a theoretical lower bound to benchmark the miss penalties of other demand-based policies. In the same way that Belady's OPT algorithm [52] is of interest for studying page replacement policies.

For any given ω, a VM management policy π is a demand-based policy if π transitions a VM V_k to the booted state at time t (if not already booted) only when

$$\exists r_j \in \omega,$$

where $r_j = (V_k, t_j, d_j)$, for a given VM V_j, some time instant t_j, and some duration d_j. For any given ω and π, we define $h_i, i \in \{0, 1, \ldots n, n+1\}$, to be the number of requests $r_j \in \omega$, such that the target VM (i.e., V_k) was in state S_i on arrival of the requests. Note that, $\sum_{i=0}^{n+1} h_i = |\omega|$.

As an example, consider a scenario where three VMs – V_1, V_2, and V_3 – are provisioned on a machine with three VM states S_0, S_1, and S_2. Let all three VMs be initially in state S_2 (shutdown), and let $\omega = r_1, r_2, r_3$, where $r_1 = (V_1, 0, 100)$, $r_2 = (V_2, 100, 100)$, and $r_3 = (V_3, 200, 100)$. In this case, $h_2 = 3$ and $h_0 = h_1 = 0$, since the target VMs for requests r_1, r_2, and r_3 are in the state S_2 upon arrival of the requests, and $\sum_{i=0}^{i=2} h_i = |\omega| = 3$.

[2]Section 5.6 presents an alternate mathematical formulation of the problem, and quantifies the space of possible idle VM management policies.

Upon the arrival of the request $r_j = (V_k, t_j, d_j)$, its target VM V_k is transitioned to the booted state from its current state. Since the maximum number of VMs in each state is limited (by B_i), this transition *may* require additional transitions to transition other VMs that are idle from: (a) the booted state into inactive states and (b) from one inactive state to another. Even if all transitions are conducted in parallel, these additional transitions, depending upon their duration, may contribute towards increasing the miss penalty. For instance, an additional transition may take more time than the time to transition the target VM to the booted state. Thus, for any demand-based VM management policy π,

$$P_\pi(\omega) \geq Min_\pi(\omega),$$

where $Min_\pi(\omega) = \sum_{i=0}^{n+1} h_i.t'_i$.

Lower bound on demand-based policies

Gill et al. [88] propose a demand-based multi-level cache management policy which provides the lowest average response time and lowest inter-cache bandwidth. Along the same lines, we define a VM management policy ϕ such that

$$\forall\pi, \forall\omega, Min_\phi(\omega) \leq Min_\pi(\omega),$$

and hence, $\forall\pi, Min_\phi(\omega) \leq P_\pi(\omega)$. That is $Min_\phi(\omega)$ forms a lower bound on the total miss penalty for any demand-based VM management policy π, for any request string ω of length $|\omega|$.

We now compute the lower bound $Min_\phi(\omega)$. First, consider a state hierarchy which consists only of the booted and shutdown states (S_0 and S_1 with $n = 0$). Consider the application of Belady's OPT algorithm on this hierarchy for managing VMs, that is, when a VM needs to be transitioned to the booted state and the number of booted VMs equals B_0, the booted idle VM which will receive a request farthest in the future is transitioned to the shutdown state. Using this algorithm on a two-state hierarchy, for any given ω, let the optimal number of hits $hOPT(\omega, B_0)$ be the number of requests in ω such that the target VM is in the booted state (S_0) on arrival of the request, where B_0 is the maximum number of VMs possible in the booted state.

Let ϕ be a demand-based VM management policy which, for a reference string ω, for each state S_i ($i \in 0, 1, \ldots n$), exhibits h_i (number of hits to state S_i), such that,

$$h_i = hOPT(\omega, \sum_{j=0}^{i} B_j) - hOPT(\omega, \sum_{j=0}^{i-1} B_j). \tag{5.1}$$

We show that for all demand-based policies, ϕ minimizes $Min_\pi(\omega)$, and hence $Min_\phi(\omega)$ forms the lower bound on the total miss penalty of any demand-based policy.

Lemma I

For all demand-based policies, policy ϕ maximizes $\sum_{i=0}^{k} h_i, \forall k \in \{0, 1 \ldots n\}$.
Proof: Summing (5.1) over the range $i = 0, 1 \ldots k$ we get,

$$\sum_{i=0}^{k} h_i = hOPT(\omega, \sum_{j=0}^{k} B_j).$$

This is the same as operating Belady's OPT algorithm on a hierarchy with just two states – booted and shutdown, with the maximum possible number of VMs in booted equal to $\sum_{j=0}^{k} B_j$. Since Belady's algorithm is known to be the optimal demand-based policy on a two-state hierarchy, $\sum_{i=0}^{k} h_i$ is maximized.

Lemma II

For any ω, no other demand-based policy π has $Min_\pi(\omega) < Min_\phi(\omega)$.
Proof: We prove by contradiction. Let $\widehat{\pi}$ be a demand-based policy (with respective $\widehat{h_i}$), such that $Min_{\widehat{\pi}}(\omega) < Min_\phi(\omega)$. Therefore,

$$\sum_{i=0}^{n+1} \widehat{h_i}.t'_i < \sum_{i=0}^{n+1} h_i.t'_i$$

$$\text{Or, } \sum_{i=0}^{n} \widehat{h_i}.(t'_i - t'_{n+1}) + (\sum_{i=0}^{n+1} \widehat{h_i}).t'_{n+1} < \sum_{i=0}^{n} h_i.(t'_i - t'_{n+1}) + (\sum_{i=0}^{n+1} h_i).t'_{n+1}.$$

Since $\sum_{i=0}^{n+1} \widehat{h_i} = \sum_{i=0}^{n+1} h_i = |\omega|$,

$$\sum_{i=0}^{n} \widehat{h_i}.(t'_{n+1} - t'_i) > \sum_{i=0}^{n} h_i.(t'_{n+1} - t'_i). \tag{5.2}$$

$$\text{Or, } \sum_{i=0}^{n} \widehat{h_i}.(t'_n - t'_i + t'_{n+1} - t'_n) > \sum_{i=0}^{n} h_i.(t'_n - t'_i + t'_{n+1} - t'_n).$$

$$\text{Or, } \sum_{i=0}^{n} \widehat{h_i}.(t'_n - t'_i) > \sum_{i=0}^{n} h_i.(t'_n - t'_i) + (\sum_{i=0}^{n} h_i - \sum_{i=0}^{n} \widehat{h_i}).(t'_{n+1} - t'_n).$$

The second term on right hand side is non-negative because $t'_{n+1} \geq t'_n$, and by Lemma I, $\sum_{i=0}^{n} h_i \geq \sum_{i=0}^{n} \widehat{h_i}$. This means,

$$\sum_{i=0}^{n} \widehat{h_i}.(t'_n - t'_i) > \sum_{i=0}^{n} h_i.(t'_n - t'_i).$$

Since the nth term in the summation on either side is zero, we get,

$$\sum_{i=0}^{n-1} \widehat{h_i}.(t'_n - t'_i) > \sum_{i=0}^{n-1} h_i.(t'_n - t'_i). \tag{5.3}$$

In reducing (5.2) to (5.3), the summation reduces from n to $(n-1)$. Since,

$$\forall j \in \{1, \ldots n\}, t'_{j-1} \leq t'_j.$$

Therefore, these steps can be repeated until the summation reduces to $n = 1$. That is,

$$\widehat{h_0}.(t'_1 - t'_0) > h_0.(t'_1 - t'_0),$$

which implies $\widehat{h_0} > h_0$, which contradicts Lemma I (which states that ϕ maximizes $\sum_{i=0}^{k} h_i, \forall k \in \{\mathbf{0}, 1, \ldots n\}$).

Note that the lower bound $Min_\phi(\omega)$ assumes future knowledge and cannot be implemented in an online fashion. Nevertheless, it serves as theoretical lower bound for comparing with other demand-based policies. In Section 5.5, we compare a few widely used demand-based policies with the lower bound and compare them with proactive policies.

5.2.2 Proactive Policies

Proactive policies configure, transition, or provision a system resource prior to a demand for it being received. They have previously been explored in the context of page replacement, with the goal of producing lower page faults than demand-based page replacement. Existing work [176] has shown that Belady's OPT algorithm is the optimal demand-based policy that minimizes the number of page fetches but does not minimize the number of page faults, because it does not prefetch pages. Trivedi et al. [176] explored the use of proactive policies to lower the number of page faults, and proved that DPMIN [176] is the optimal pre-paging algorithm. DPMIN proceeds as follows: at the time of a page fault, DPMIN scans the future page reference string and fetches the first m pages that will be referenced in the future (including the page that caused the page fault), where m is the number of memory page frames. For the purposes of this work, we define *proactive policy* as any policy that is not a demand-based policy (as defined in Section 5.2.1).

SlidingWindow

We extend the technique used in the DPMIN algorithm to define the SlidingWindow policy for managing VMs across different states. Our rationale is that, since VMs can be transitioned in parallel, provisioning VMs in anticipation of requests (e.g., when servicing another VM's request) should reduce average miss penalties.

SlidingWindow proceeds as follows: whenever a request $r_t \in \omega$ is received such that the target VM is not in the booted state, it computes a new state configuration for all VMs. To compute this new configuration, SlidingWindow scans the future reference string (illustrated in Figure 5.2). All VMs that are currently active are retained in their booted state in the new configuration. If A is the number of currently active VMs, the first $(B_0 - A)$ VMs (that are currently idle) that will be requested in the future are placed into the booted state (S_0) in the new configuration (illustrated as a time window W_0). Similarly, the B_1 VMs that will be requested next are placed into S_1 (illustrated as time window W_1). This process continues up to state S_n (time window W_n) and the remaining $(v - \sum_{i=0}^{n} B_i)$ VMs are placed in the shutdown state. After the new configuration is computed, VMs are transitioned from their existing to this new configuration.

When VM V_i becomes idle, SlidingWindow re-scans the future reference string to compute t_{next}, that is, the time at which V_i will be requested next. If t_{next} falls in the time window W_j it transitions V_i into S_j, and one VM from S_k to S_{k-1} (the one that is refer-

enced the soonest) $\forall k \in \{j, j-1, \ldots 1\}$. In effect, it slides the windows $W_0, W_1 \ldots W_{j-1}$ to the right. If t_{next} falls in the window W_0, V_i remains in the booted state.

Similar to DPMIN, our SlidingWindow policy assumes knowledge of the future reference string, and cannot be implemented in an online fashion. Therefore, in addition to the offline implementation of the policy, we provide an online implementation (called *SlidingWindow+ARMA*) which uses a predictor to estimate t_{next}, and uses the predicted value to perform its proactive VM provisioning. We describe SlidingWindow+ARMA in detail in Section 5.4.2 and compare it with demand-based policies in Section 5.5.

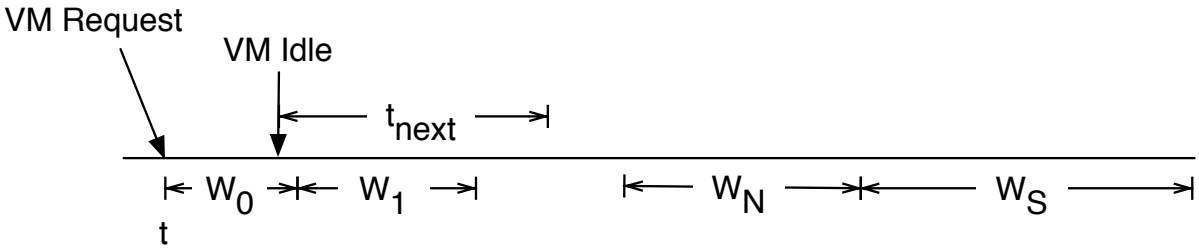


Figure 5.2: Proactive idle VEE management using the SlidingWindow policy.

5.3 Obtaining Model Parameters

Our formal model (described in Section 5.2) relies on a few input parameters, namely the matrix of transition times ($T_{(n+2)\times(n+2)}$) and the maximum number of VMs that can be simultaneously hosted in each state (B_i). We use LXC [22], a widely used OS-level virtualization solution, as an example virtualization solution, and conduct an experimental analysis to obtain these model parameters. Note that we continue to refer to execution environments created using LXC as VMs, since our formal model (Section 5.2) is applicable to any virtualization solution. We now describe our methodology in detail and justify our choices.

5.3.1 LXC as a Case Study

OS-level virtualization approaches have been shown to incur 40-50% lower virtualization overhead than other approaches such as Xen paravirtualization [143, 172], thus

promising potentially higher density. Example OS-level virtualization solutions include LXC [22], OpenVZ [28], and VServer [23]. We choose LXC as our example virtualization solution for several reasons:

(i) it is open source, allowing easy analysis of its implementation,

(ii) it is in production use and is part of the mainstream Linux kernel,

(iii) it offers a low-latency inactive VM state called "frozen",

(iv) it is being used in other projects which can benefit from increased VM density, such as Docker [9] (to provide "frozen in state apps"), and Confidential Commuting [77] (to provide per-user private VEEs).

As noted, in addition to the booted (S_0) and shutdown state (S_2), LXC implements a *frozen* state (S_1) which forms a middle ground between booted and shutdown states. When an idle VM in the booted state is transitioned to the frozen state, it retains its memory and disk footprint, while relinquishing CPU cycles. In addition, frozen-to-booted transition times are significantly smaller than shutdown-to-booted transitions. Thus, LXC provides us with a three-state hierarchy with the booted, frozen, and shutdown states. Menage [133] provides a detailed description of the implementation of the frozen state.

5.3.2 Experimental Setup

We use LXC [22] (v.0.8.0) to create VMs. VMs are hosted on a machine that has four Intel Xeon processors with six 3.46 GHz cores each and 128 GB of RAM. It contains a 7200 RPM 1 TB SATA hard disk that we use to store VMs' OS images. All experiments are repeated 50 times and all results are reported using averages with 95% confidence intervals.

VMs in an OS-level virtualization solution, such as LXC, share the host kernel's process pool, data structures, and devices. Therefore, when increasing VM density, some host kernel parameters need to be increased. For instance, since all processes of all LXC VMs share the host kernel's pool of open file descriptors (FD) the total number of open FDs allowed by the host kernel limits the number of VMs. Similarly, the kernel's maximum allowed process identifier (PID) (set to a default of 32,767), and the number of Unix98 pseudoterminals (set to a default of 4,096), also limit the number of VMs. For conducting our experiments we increased these values to 14,000,000, 65,535, and 8,000 respectively.

5.3.3 Quantifying Density

We first determine the maximum number of booted, frozen, and shutdown VMs that can be supported in our testbed.

Shutdown VMs

The only system resource consumed by shutdown VMs is disk space. It is required for storing their OS image, applications, and libraries. In our testbed, each VM consumes 476 MB of disk space. Thus, 2,000 VMs can be created on a 1 TB disk, using the EXT4 filesystem.

Booted VMs

To determine the maximum number of booted VMs (B_0) we conduct a simple experiment where the number of booted but idle VMs on the machine is gradually increased, while all other VMs are shutdown. VMs are transitioned from shutdown to booted sequentially, with a delay of 30 seconds between successive VMs to ensure that the system reaches a steady state; essential for recording system measurements. Measurements are recorded using `vmstat`.

We observe that the system memory consumption increases steadily with increasing number of VMs. This is because each additional VM consumes approximately 42 MB of memory for initializing its processes (such as its SSH server, and other daemons). Figure 5.3 shows the CPU system time (time spent running kernel code), and CPU idle time averaged over all 24 cores on the server machine, as the number of booted idle VMs is increased. We observe that up to 250 VMs the CPU is largely idle. This is because idle VMs do not perform any significant computation and only a few VM processes (such as udevd, and other daemons) are running, this slightly decreases the CPU idle time, while other processes remain blocked. Note that, all VM processes are in user space. However, beyond 250 VMs, we observe an increase in CPU system time, eventually reaching 100% at 450 VMs. At this stage, no additional VMs can be booted as all 24 cores are completely busy. We believe that this is because of the inability of LXC's cgroup handler [133] to scale with the increasing number of processes. LXC uses the cgroup handler for the bookkeeping of resources used by processes of different VMs and to enforce isolation. As a result, this limits the number of booted idle VMs to approximately 250.

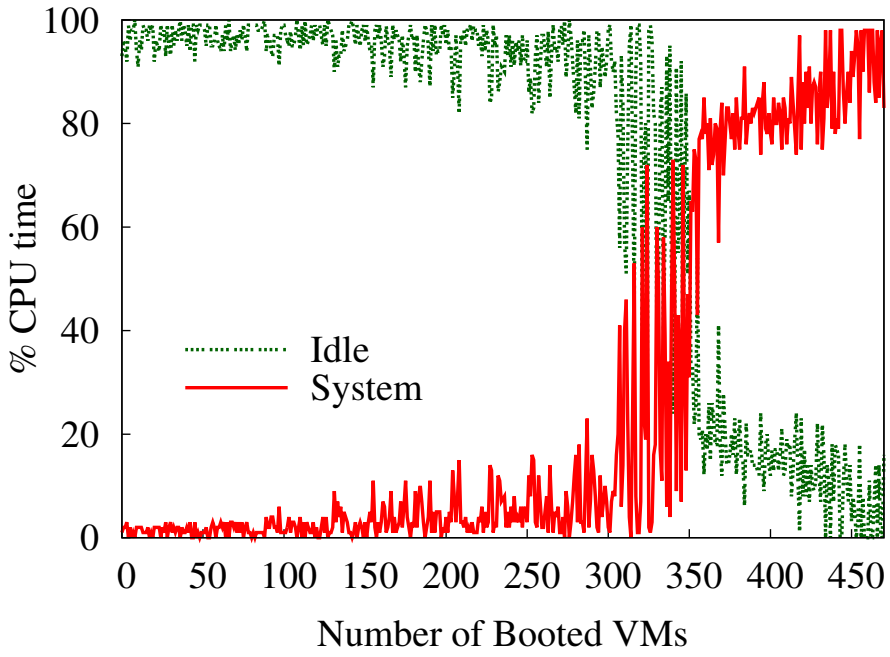


Figure 5.3: CPU utilization versus the number of booted VMs.

Frozen VMs

We determine the maximum number of frozen VMs (B_1) by conducting a simple experiment where the number of frozen VMs on the machine is gradually increased. Each VM is booted, allowed to initialize, and then transitioned to frozen. To determine when a VM has finished booting, we use `netcat` to detect if the VM's SSH server has started. We measure the time taken to boot-up the VM and start its SSH server. All other VMs remain in the shutdown state.

We observe a steady increase in the system memory consumption (at approximately 42 MB per VM). This is because processes in a frozen VM retain their memory footprint (due to the absence of memory pressure). Moreover, we observe that the time to transition a VM from shutdown to booted (before transitioning to frozen) increases considerably as the number of frozen VMs on the machine increases. This is because many system calls used by LXC for booting a VM, such as `fork`, `wait`, and `open`, take more time to complete, as the number of frozen VMs increases. As described in Section 5.3.4, this increase in transition time eventually limits the number of frozen VMs (to approximately 300 frozen VMs).

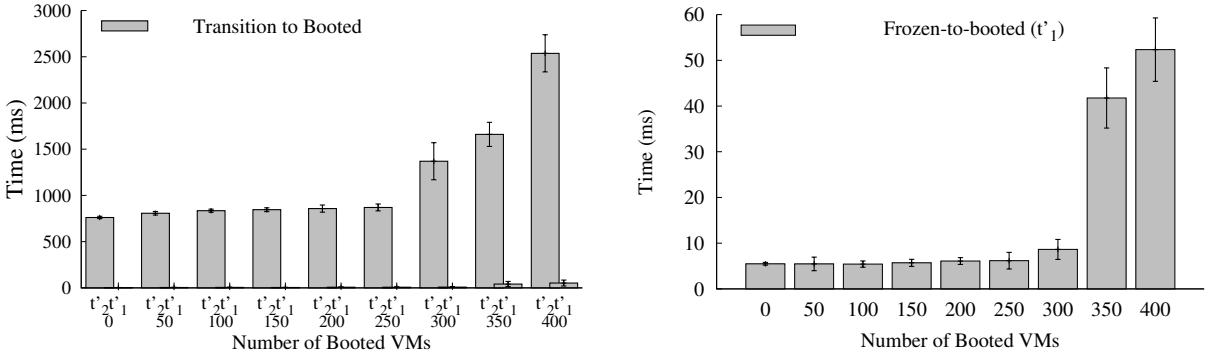
5.3.4 Impact of Density on Transition Time

We study the effect of VM density in each state on the transition times. We vary the number of booted VMs and measure the different state transition times (while keeping the number of frozen VMs at zero). Similarly we vary the number of frozen VMs and measure the transition times (while keeping the number of booted VMs at zero).

Varying the Number of Booted VMs

The number of booted VMs is increased in steps of 50. At each step the different transition times are measured (for a given VM). All other VMs are in the shutdown state.

Figure 5.4a shows the shutdown-to-booted (denoted t'_2) and frozen-to-booted (denoted t'_1) transition times. We observe that up to 250 VMs, the shutdown-to-booted transition time remains largely constant. However, beyond 250 booted VMs we see a considerable increase in boot time. This is due to a surge in CPU system time at 300 VMs and beyond (as explained in Section 5.3.3, Figure 5.3), which reduces available CPU time to zero. Figure 5.4b shows the frozen-to-booted transition time (denoted t'_1 in Figure 5.4a), with a magnified time axis. We observe that frozen-to-booted transition times



(a) Transition times for Shutdown-to-Booted (t'_2) and Frozen-to-Booted (t'_1).

(b) Transition times for Frozen-to-Booted (t'_1).

Figure 5.4: Transition times with increasing number of booted VMs.

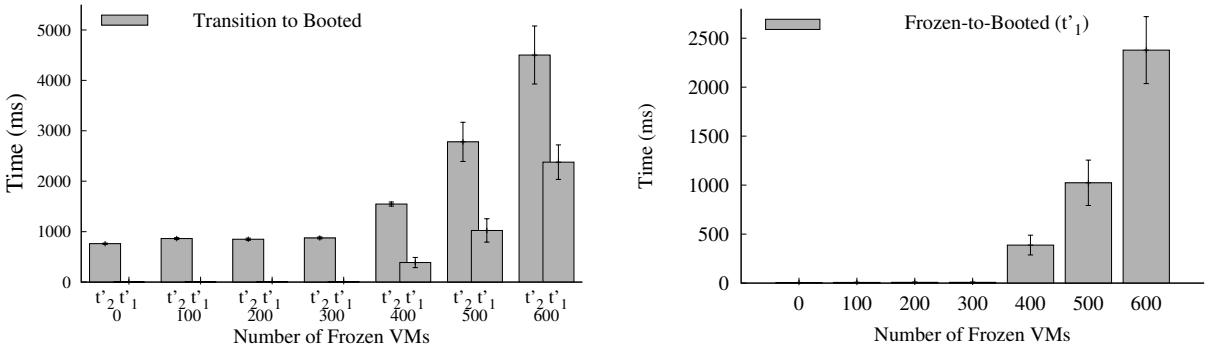
increase only slightly with increasing number of booted VMs and is considerably smaller than shutdown-to-booted transition time. This is because frozen-to-booted is a comparatively faster and less resource-intensive transition. The abrupt rise beyond 300 booted VMs, is attributed to the surge in CPU system time, as explained in Section 5.3.3. Along the same lines we also measure the booted-to-shutdown ($t_{0,2}$) and booted-to-frozen ($t_{0,1}$) transition times while increasing the number of booted VMs. However, we do not observe any significant variation in these transition times, remaining constant at averages of 640 ms and 0.15 ms respectively.

Varying the Number of Frozen VMs

In a similar fashion to the previous experiment, the number of frozen VMs is gradually increased and the different transition times are measured.

Figure 5.5a shows the shutdown-to-booted (denoted t'_2) and frozen-to-booted (denoted t'_1) transition times. We observe that the time for a shutdown-to-booted transition increases only slightly up to 300 frozen VMs and is considerably larger beyond that point. This is because, as discussed in Section 5.3.3, system calls used by LXC for booting, require more time to complete when the number of frozen VMs increases, leading to increased transition times. Figure 5.5b shows the frozen-to-booted transition times (denoted as t'_1 in Figure 5.5a) with a magnified time axis. We observe that below 300 VMs, the transition time is largely constant. However, beyond 300 frozen VMs, the transition

time increases considerably. This is because LXC's cgroup freezer mechanism [133] begins consuming more time for unfreezing a VM, and hence does not scale. Therefore, the number of frozen VMs is limited by this increased transition time, to approximately 300. As in the previous experiment, we find that the booted-to-shutdown ($t_{0,2}$) and booted-to-frozen ($t_{0,1}$) transition times do not change significantly with the number of frozen VMs. These transitions times are measured at averages 642 ms and 0.17 ms respectively, which is nearly identical to their values in case of varying number of booted VMs.



(a) Transition times for Shutdown-to-Booted (t'_2) and Frozen-to-Booted (t'_1).

(b) Transition times for Frozen-to-Booted (t'_1).

Figure 5.5: Transition times with increasing number of frozen VMs.

5.3.5 Deriving the Model Parameters

We now describe how we derive the parameters for our model (in Section 5.2) using the experimental analysis (described above).

Due to increase in the transition times as the number of booted VMs is increased (as explained above, Figures 5.3, 5.4), we define the maximum number of VMs possible in the booted state (B_0) to be 250. Similarly, given the variation in transition times as the number of frozen VMs is increased (Figure 5.5), the maximum number of VMs possible in the frozen state (B_1) is 300. Note that both B_0 and B_1 values are derived from limitations in LXC's design and implementation (described above). While it may be interesting to explore these limitations in greater detail (and alleviate them), such work lies outside the scope of our current work.

We compare the shutdown-to-booted (t'_2) and frozen-to-booted (t'_1) transition times in the two sensitivity analysis experiments (i.e., Figure 5.4a and Figure 5.5a). Interestingly, we find that within the operating range of up to 250 booted VMs, and up to 300 frozen VMs, the respective transition times in either experiments differ insignificantly. For instance, shutdown-to-booted transition time when varying only the number of booted VMs (up to 250 booted VMs, Figure 5.4a), is comparable to shutdown-to-booted transition time when varying only the number of frozen VMs (up to 300 frozen VMs, Figure 5.5a). Other transitions times (frozen-to-booted $t_{1,0}$, booted-to-frozen $t_{0,1}$, and booted-to-shutdown $t_{0,2}$) exhibit similar behaviour. Thus, we believe that within this operating range (up to 250 booted and 300 frozen VMs) all transition times remain largely constant, *regardless of the number of booted and frozen VMs*. We therefore represent the average transition times (within the operating range) using the transition matrix $T_{3\times3}$ shown in Table 5.1, where $t_{i,j}$=time to transition from state S_i to S_j. Note that to transition a frozen LXC VM to shutdown (and vice versa), it must first be transitioned to booted.

From \ To	Booted (S_0)	Frozen (S_1)	Shutdown (S_2)	B_i
Booted (S_0)	0.00 ms	0.17 ms	641.64 ms	250
Frozen (S_1)	4.43 ms	0.00 ms	-	300
Shutdown (S_2)	802.16 ms	-	0.00 ms	2000 (Disk limited)

Table 5.1: Transition matrix ($T_{3\times3}$) and state capacity (B_i) values for LXC.

5.4 Policy Comparison Setup

Using the parameters obtained above we study the effect of different policies using a simulation-based analysis, which makes exploring a large design space feasible.

In this section, we first discuss the design and implementation of our simulator, followed by the description of the policy implementations in Section 5.4.2, the personal VEE workloads we use for comparing policies in Section 5.4.3, and our evaluation metric in Section 5.4.4.

5.4.1 Simulator Design and Implementation

Our simulator consists of a *policy module*, a *cost-model module*, and a *workload module*. The cost-model module encapsulates the VM hierarchy parameters (B_i and $T_{3\times3}$). The policy module encapsulates all logic pertaining to a particular policy and maintains any in-memory state required for implementing that policy such as any per-VM bookkeeping. The workload module encapsulates all logic required for simulating the VMs' workloads, such as distributions for request inter-arrival times and request durations. Such modularity allows us to easily extend the simulator to study different workloads, different VM management policies, as well as different VM hierarchies, by simply implementing the respective module. The simulator maintains the current state assignment for each VM. In addition, it contains a single time-sorted event queue and a single thread which processes events in this queue. Events are either of type *VMrequest* or *VMidle*.

At initialization time, the workload module generates the simulated VMrequest events for the required number of VMs and populates the event queue. When processing a VMrequest event, the simulator passes the current state assignment, the request event, and event queue to the policy module, and receives the updated state assignment. It then compares the updated state assignment with the current one and computes a list of required VM transitions. It then computes the time required to perform the transitions, updates the state assignment, and enqueues a VMidle event with a later timestamp (derived using the request duration in the VMrequest event) into the event queue. Note that all invocations of the policy module are serialized. For instance, if a VMrequest event occurs while the policy module is computing the updated VM state assignment, it is processed after the policy module finishes its computation. Lastly, when computing the time taken to perform a set of transitions, the simulator assumes that different VMs' transitions take place in parallel. This assumption is justified because in doing so we are able to measure the best case behavior of any policy. Any additional overhead in the system in performing parallel transitions would only serve to increase the transition times, and the miss penalty so obtained would still be bounded by that in the best case scenario. VMidle events are processed in the same fashion as VMrequest events. However, some policies (e.g, demand-based policies) may choose to not take any action when a VM becomes idle.

Implementation

We have implemented the simulator using C# over the .NET v4.5 framework. We have implemented policy modules for different reactive and proactive policies, the workloads

described in the next section, and a cost-model module for LXC. We validate the simulator using sample deterministic workloads and state hierarchies. The implementation is publicly available at http://github.com/rayman7718/VMSim.

5.4.2 Policy Implementations

We implement the following idle VEE management policies.

LRU

Least Recently Used (or LRU) [70] is a policy that is widely studied for page replacement. We apply it to idle VM management in a cascaded fashion. For each VM, it maintains the timestamp of the last request (t_r). All VMs are initially in the shutdown state. As requests for different VMs arrive, they are transitioned to the booted state. Eventually, as the number of VMs in the booted state (i.e., including active and idle VMs) reaches the limit B_0, for each VM transitioning into booted thereafter, LRU chooses to transition the booted VM with the minimum t_r into the frozen state. In effect, for each VM it uses the time since the last request to estimate the likelihood that it will be requested again. Similarly, as the number of VMs in the frozen state reaches B_1, for each VM transitioning into frozen thereafter, the frozen VM with the minimum t_r is shut down. Note that, unlike traditional implementations of LRU (e.g., in page replacement) where all timing information of the resource is deleted after its eviction, we maintain t_r for each VM after eviction in order to apply it across multiple states.

Cascaded Belady's OPT

This policy is simply an extension of Belady's OPT algorithm to multiple states applied to idle VM management. That is, when number of VMs in any state S_i exceeds B_i, the VM that is referenced the furthest in the future is transitioned to S_{i+1}. This version of Belady's OPT algorithm is known to be suboptimal [88]. However, we implement it in order to compare it with demand-based policies that have no future knowledge (e.g., LRU) and the lower bound $Min_\phi(\omega)$. To the best of our knowledge such a comparison has not been conducted in previous work.

Lower Bound on demand-based policies

As explained in Section 5.2.1, $Min_{\phi}(\omega)$ forms the lower bound on demand-based policies. We first obtain the h_0, h_1, h_2 values for LXC as per Eq. 5.1 for the different workloads we study (i.e., different ω values). For each ω, we obtain $h_0 = hOPT(\omega, B_0)$, $h_1 = hOPT(\omega, B_0 + B_1) - hOPT(\omega, B_0)$, and $h_2 = |\omega| - h_0 - h_1$. We then determine the lower bound $Min_{\phi}(\omega) = (h_0.t'_0 + h_1.t'_1 + h_2.t'_2)$.

SlidingWindow+GroundTruth

As explained in Section 5.2.2, our SlidingWindow policy requires knowledge of the future. Therefore, we implement the offline version of SlidingWindow which uses knowledge of the future (called *SlidingWindow+GroundTruth*).

SlidingWindow+ARMA

We also provide an online implementation of the SlidingWindow policy (called *SlidingWindow+ARMA*) which attempts to predict the inter-arrival time for each VM and updates the prediction model at the time of each request for that VM. We employ the widely-used *auto regressive moving average* (ARMA) time-series model [134] for predicting the inter-arrival times. To find the order of the ARMA model we employ Bayesian information criterion [160] and at each model update we perform a maximum likelihood fit on the inter-arrival times. The policy then uses the predicted inter-arrival time for each VM to perform its proactive actions. Therefore, its miss penalty depends directly on the prediction error of the ARMA model.

5.4.3 Workload Analysis

We analyze personal VEE workloads by categorizing them into three categories based on the requests' inter-arrival and duration times. We believe that such categorization (as opposed to a mixed workload) allows us to better understand the behavior of different policies.

Fixed Inter-arrival Time, Fixed Duration

A common use of personal VEEs is to periodically upload a fixed amount of data from in-home sensors (e.g., from energy sensors, as described in Chapter 3) or a user's smartphone [77, 137]. In these scenarios, requests have relatively fixed inter-arrival times and durations. For such requests both the arrival and departure of requests are highly predictable and form an interesting point of comparison between proactive and reactive policies. Similarly tasks involving periodic pre-processing of data [158, 169] also fall under this category.

Stochastic Inter-arrival Time, Fixed Duration

Users can also use their personal VEEs to host private instances of common application servers such as web, mail, the Bolt metadata server (described in Chapter 4), and other servers [58, 163, 164]. Due to the user-facing nature of these application servers, the requests they receive have stochastic inter-arrival times, and typically involve downloading or uploading fixed amounts of data, thus leading to requests with a relatively fixed duration.

Stochastic Inter-arrival Time, Stochastic Duration

In many personal VEE applications the requests are user-generated, and thus have stochastic inter-arrival times. In addition, there are a wide variety of such requests. As a result, these requests vary in the amount of data processed and the type of computation performed which results in stochastic duration times. Examples include private data analytics applications (described in Chapter 3), VM-backed mobile applications [158], and other similar applications [107] (discussed in detail in Sections 2.1.4 and 2.1.5).

5.4.4 Metric

In order to compare different policies for a request string ω, we use the *average miss penalty* incurred by any policy π, defined as:

$$\text{Average miss penalty} = \frac{P_\pi(\omega)}{|\omega|}.$$

We choose this metric because:

(i) it captures the miss penalty across all VMs in the system,

(ii) it allows us to observe the behavior of any policy while increasing the number of total VMs, and

(iii) it allows us to easily observe the differences between reactive and proactive policies.

Note that the lower bound on average miss penalty of reactive policies is expressed as $\frac{Min_{\phi}(\omega)}{|\omega|}$.

5.5 Simulation Results

We now study the effect of increasing VM density on the average miss penalty for different policies. The request string ω in all simulations contains 100 arrivals per VM, that is, $|\omega|$=Total #VMs $\times$ 100.

In order to simulate stochastic inter-arrival times, we use the request inter-arrival times from publicly available web trace data [49], since it has been used in evaluating on-demand VM provisioning in existing work [118]. Similarly in order to simulate stochastic request durations, we use the web connection duration characterization provided by Newton et al. [141]. We believe this to be a representative request duration characterization for personal servers. Lastly, we use the respective mean values from the two datasets for generating the fixed inter-arrival time (of 50 s) and fixed duration (of 10 s) workload. Table 5.2 shows the variability of inter-arrival times and durations for the three cases we study. Note however that the policies we consider do not utilize request duration and we defer the design of policies that consider request durations to future work.

In each workload experiment (below), we start the x-axis at 250 VMs since it is the maximum number of VMs possible in the booted state (and there are no miss penalties below 250 VMs). We increase VM density to the point that the total number of simultaneously active VMs increases above 250 (equal to B_0) and cannot be hosted using this hierarchy. Note that this limit is a result of the current workload (longer idle times would increase this limit).

Case		Inter-arrival times				Durations			
Inter-arrivals	Durations	Mean	Standard deviation	Min	Max	Mean	Standard deviation	Min	Max
Fixed	Fixed	50.0 s	0.0 s	50.0 s	50.0 s	10.0 s	0.0 s	10.0 s	10.0 s
Stochastic	Fixed	50.0 s	160.5 s	10.0 s	941.2 s	10.0 s	0.0 s	10.0 s	10.0 s
Stochastic	Stochastic	50.0 s	163.0 s	0.9 s	941.2 s	10.0 s	22.4 s	5.0 ms	120.0 s

Table 5.2: Variation of request inter-arrival times and durations across the three test cases.

5.5.1 Fixed Inter-arrival Time, Fixed Duration Workload

Figure 5.6 shows the average miss penalty versus VM density for different policies, for a request string ω with a fixed inter-arrival time of 50 seconds and a fixed request duration of 10 seconds. Figure 5.6 also shows the shutdown-to-booted and frozen-to-booted transition times for comparison. For each VM, the time at which its first request is received is chosen uniformly at random from [0,50 s]. Later in this section, we analyze the behavior of the policies under other inter-arrival time and duration values.

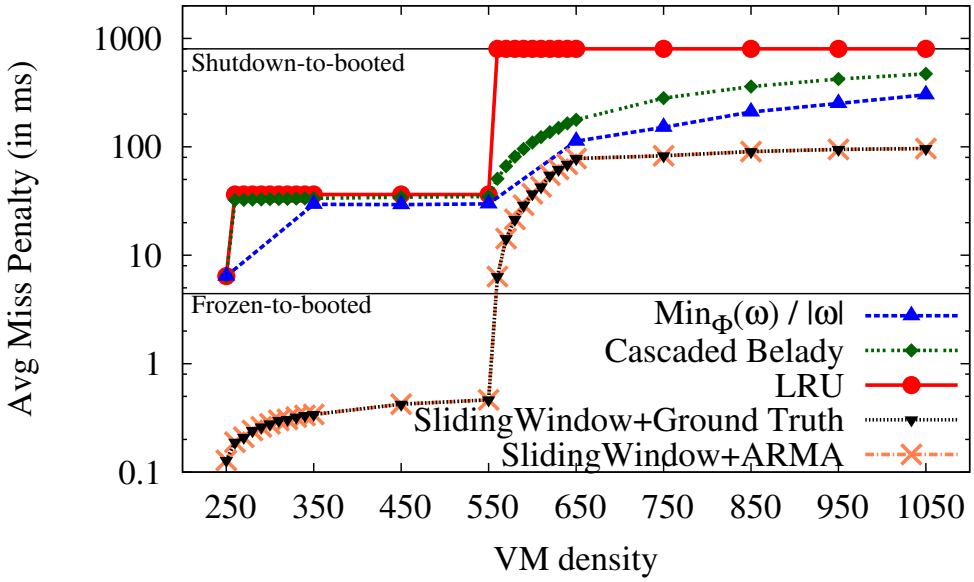


Figure 5.6: Average miss penalty with increasing VM density for fixed inter-arrival time and duration.

We first explain the behaviour of the reactive policies. We observe that at a VM density of 250, both reactive policies, LRU and Cascaded Belady's OPT, incur the same average miss penalty which matches the lower bound. This is because, in this case, all reactive policies transition each VM into the booted state once its first request is received and no idle VMs need to be transitioned out of the booted state thereafter (since VM density equals B_0). As VM density increases further, not all VMs can remain booted. Depending on the policy, some VMs are transitioned to frozen (when idle) and are transitioned to booted when their request arrives, which increases the average miss penalty. Similarly, as VM density increases beyond 550, not all VMs can be in either booted or frozen states. That is, all other VMs are transitioned to the shutdown state (when idle) and are brought into the booted state when their request arrives. This contributes to increase the average miss penalty. Since shutdown-to-booted transition times are significantly higher than frozen-to-booted (approximately 800 ms vs. 4 ms), we see a much larger increase in the average miss penalty at VM density $\geq$ 550 (than at 250 VMs). Note that LRU has a significantly higher average miss penalty than the cascaded Belady's algorithm because LRU has no knowledge of the future reference string. In the case of LRU, when VM density is greater than 550, upon the arrival of each request, the target VM is always in the shutdown state thus incurring the maximum miss penalty. This is because, for this workload, between two consecutive requests to any VM V_i, there are (v-1) requests for other VMs (i.e., one per VM). Since LRU evicts the least recently used VM from booted to frozen and frozen to shutdown, when $v > 550$, V_i will get transitioned to shutdown after the intermediate (v-1) requests are serviced.

We now explain the behavior of proactive policies. Due to easily predictable fixed request inter-arrival times and durations, the average miss penalty of the SlidingWindow policy+ARMA equals that of SlidingWindow+Ground Truth. We observe that both policies incur a significantly lower average miss penalty than the reactive policies. This is because whenever a request whose target VM is not in the booted state is received, in addition to transitioning that VM to booted, SlidingWindow also transitions other VMs (as many as possible) to booted and frozen states (as explained in Section 5.2.2). Note that the proactive transitions in the SlidingWindow policies are triggered by a request whose target VM is not in the booted state. Therefore, as VM density increases up to 550, VMs span the booted and frozen states and beyond 550, VMs span all the three states. This increases the average miss penalty (since the shutdown-to-booted transition time is significantly larger than the frozen-to-booted time). Moreover, the number of idle VMs that the SlidingWindow policies can proactively transition to booted, depends on the number of active VMs at that instant (since the number of active booted + the number of idle booted = 250). As VM density increases, the number of simultaneously

active VMs increases and hence the number of VMs that can be proactively transitioned to booted decreases. This reduction in the number of possible proactive transitions also contributes to an increase in the average miss penalty.

For this workload, comparing the two online (implementable) policies (LRU and SlidingWindow+ARMA), we conclude that SlidingWindow+ARMA incurs the lower average miss penalty. It increases VM density from 250 to 550 (a factor of more than 2.2), while keeping average miss penalty under 1 ms, for a fixed inter-arrival time of 50 seconds and fixed duration of 10 seconds. Note that each VM is active for 10 seconds, then becomes idle for 40 seconds before being activated again, that is, a *mean duty-cycle* of 20%. If the maximum number of frozen VMs was not limited to 300 by LXC's implementation, for up to 250 simultaneously active VMs, a maximum of $\frac{250}{0.2}$ or 1250 VMs could be hosted using this state hierarchy while keeping the average miss penalty under 1 ms. Similarly, for an idle time of 10 seconds and active time of 40 seconds (i.e., a duty cycle of 0.8), the maximum VM density with an average miss penalty under 1 ms, would be $\frac{250}{0.8}$ or 312 VMs. To generalize, maximum VM density, for an average miss penalty ≤ 1 ms with maximum number of simultaneously active VMs ≤ 250, equals $\text{Min}\left(B_0 + B_1, \frac{B_0}{\text{mean duty-cycle}}\right)$.

5.5.2 Stochastic Inter-arrival Time, Fixed Duration Workload

Figure 5.7 shows the average miss penalty versus VM density for different policies, for a request string ω with stochastic inter-arrival time (as described above) and a fixed request duration of 10 seconds. Figure 5.7 also shows the shutdown-to-booted and frozen-to-booted transition times for comparison.

We observe that the behaviour of reactive policies in this case is similar to that in the case of fixed inter-arrival fixed duration workloads (as described above in Section 5.5.1). That is, their average miss penalty is equal and lowest at VM density of 250 VMs and thereafter it increases with VM density (and remains higher than the lower bound). Miss penalty is significantly higher at 550 VMs (and higher) than that at 250-550 VMs. This is because with more thatn 550 VMs, they span the booted, frozen, and shutdown states and shutdown-to-booted transition times are significantly larger than frozen-to-booted times.

We now explain the behaviour of the proactive policies. Due to reasons explained above in Section 5.5.1, the average miss penalty of SlidingWindow+Ground Truth increases as VM density increases. The increase is large at VM density of more than 550

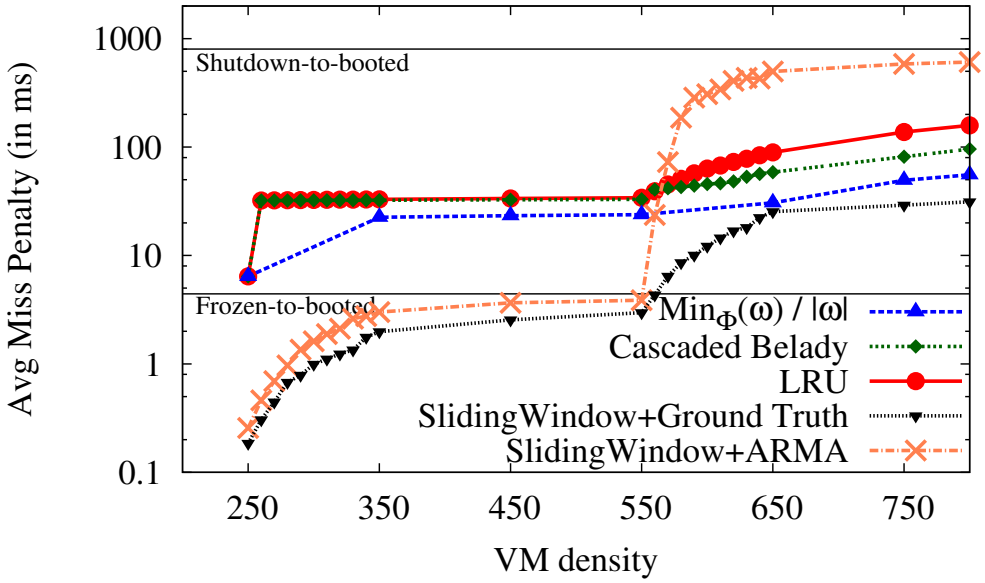


Figure 5.7: Average miss penalty with increasing VM density for stochastic inter-arrival time and fixed duration.

VMs due to the shutdown-to-booted transition time being significantly larger than the frozen-to-booted transition time. SlidingWindow+ARMA incurs a higher average miss penalty than SlidingWindow+GroundTruth. This is because SlidingWindow+ARMA uses the predicted inter-arrival time for each VM to make proactive transition decisions. Thus errors in prediction cause some VMs to transition to sub-optimal states, which increases the average miss penalty. We observe significantly larger average miss penalties as VM density exceeds 550 VMs. This is observed despite the normalized root mean square error of the prediction (shown in Figure **??**) remaining largely constant (at approximately 0.04) with increasing VM density. This is because beyond 550 VMs, they span all three booted, frozen, shutdown states (as compared to only booted and frozen below 550) and the shutdown-to-booted, booted-to-shutdown transition times are significantly higher than the frozen-to-booted, frozen-to-shutdown transition times respectively. Hence, as the number of VMs is increased, for a relatively constant degree of mis-predictions, the number of VMs that get transitioned sub-optimally to the shutdown state due to a mis-predicted inter-arrival time increases. It is the large transition times to and from the shutdown state that cause the significant increase in the average miss penalty. Due to this increase, the average miss penalty of SlidingWindow+ARMA

exceeds that of reactive approaches (beyond 560 VMs).

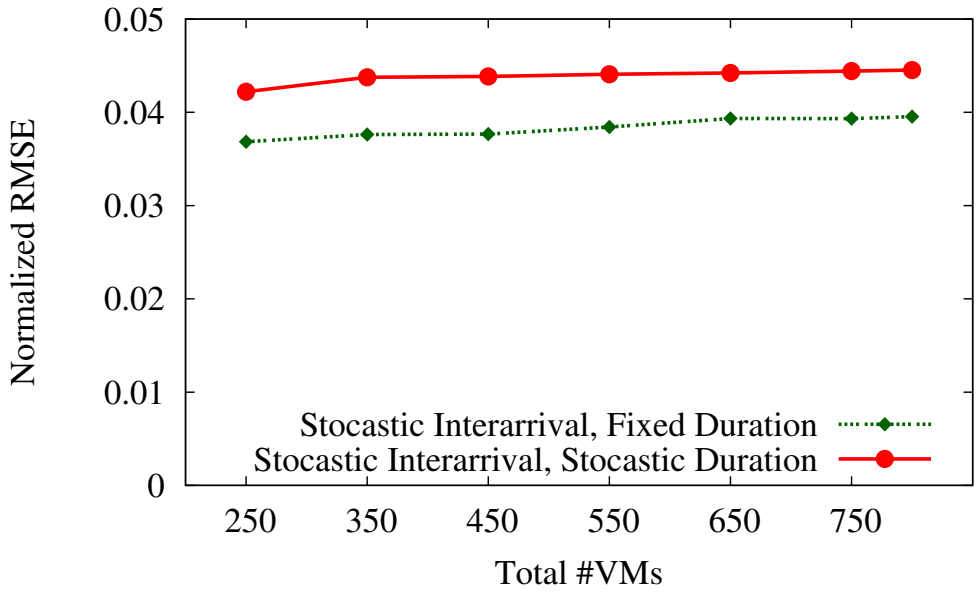


Figure 5.8: Normalized root mean square error (NMRSE) for the ARMA predictor used by SlidingWindow+ARMA.

For this workload, when comparing the two online policies, LRU and SlidingWindow+ARMA, we conclude that for a desired VM density of up to 550 VMs (2.2 $\times$ the density of current solutions), SlidingWindow+ARMA incurs a much lower average miss penalty (less than 4 ms). However, for all VM density levels larger than 560, LRU outperforms SlidingWindow+ARMA. At these density levels, some idle VMs need to be in the shutdown state (which has large transition times) and any error in inter-arrival time prediction causes a significant increase in the average miss penalty.

5.5.3 Stochastic Inter-arrival Time, Stochastic Duration Workload

Figure 5.9 shows the average miss penalty for a request string ω with stochastic inter-arrival time and stochastic duration, for different policies with increasing VM density. Figure 5.9 also shows the shutdown-to-booted and frozen-to-booted transition times for comparison.

We observe that the stochastic request duration has little effect on the average miss penalty, which is similar to the case of stochastic inter-arrival time and fixed duration (described above in Section 5.5.2). We believe that this is because both the reactive and proactive policies we study (LRU, Cascaded Belady's algorithm, and SlidingWindow), only use request inter-arrival time in making their reactive or proactive transition decisions. An interesting avenue for future work would be to formulate policies which also take into account the expected request durations. The behaviour of our current policies can be explained using reasoning similar to that in Section 5.5.2.

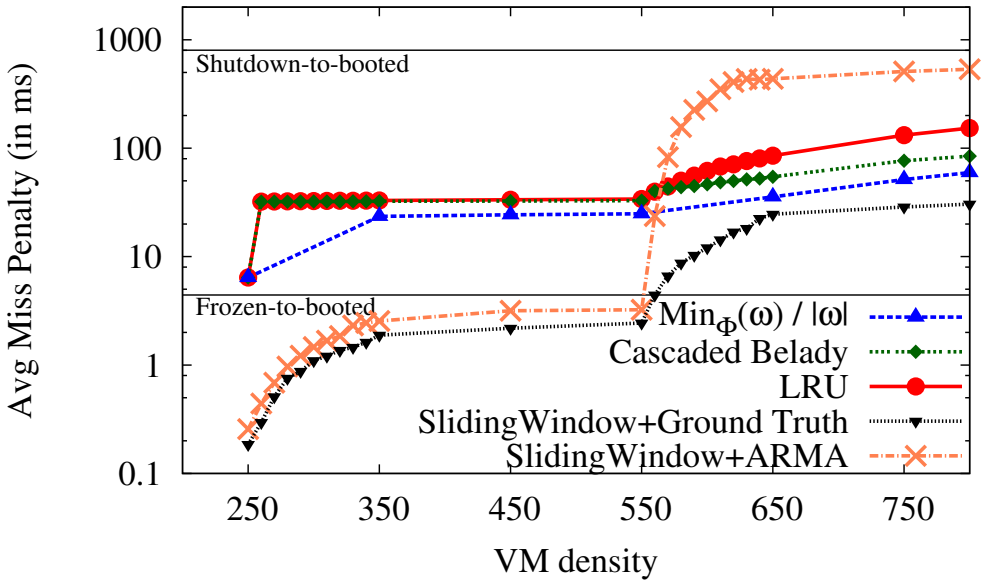


Figure 5.9: Average miss penalty with increasing VM density for stochastic inter-arrival time and stochastic duration.

5.5.4 Summary of Simulation Results

The key findings of the simulation-based policy comparison can be summarized as follows:

- Online proactive policies such as SlidingWindow+ARMA are highly sensitive to prediction error. Thus for any given hierarchy, to ensure low miss penalties, such policies should be used only when the desired VM density is such that all VMs can

be accommodated within inactive states with low transition times. That is, when the penalties for mis-predictions are relatively small. Under all other conditions (e.g., larger VM density levels), reactive and stateless policies such as LRU should be used.

- Miss penalties of proactive and reactive (online or offline) policies, which only make use of request inter-arrival times, are largely unaffected by request durations being fixed or stochastic.
- For the given workload, using an online per-VM predictor based proactive policy results in an increase in VM density by up to a factor of 2.2, with an average miss penalty of less than 1 ms.

5.6 Characterizing the Policy Space

In Section 5.2, we formulated the idle VEE management problem. Although that problem formulation allowed us to establish a lower bound on the miss penalties of demand-based policies, it does not provide insight into finding the optimal policy. We believe that it is due to the large extent of possible policies that finding the optimal policy is relatively difficult. Therefore, in this section, we present an alternate mathematical formulation of the problem, which provides a mathematical definition for policies, and allows us to characterize the policy space.

Let $V = \{V_1, V_2, \ldots V_v\}$ denote the set of v VMs that are multiplexed on a given physical machine. Each VM is active for certain periods of time, for example, when actively serving client requests and is idle for other periods of time. We model the active and idle periods over a given time period T by a *reference string of tuples*. A *reference string of tuples* $\omega = r_1, r_2, \ldots r_T$ is a string of tuples r_i. Each r_i is a tuple $(Type, ID)$, where $Type \in \{Activate, Inactivate\}$ and $ID \in V$, which denotes the activation (i.e., start of an active period) or in-activation (i.e., start of an idle period) of the target VM V_{ID}.

For any tupled reference string ω, for each VM $V_x \in V$, if $\exists\ r_j = (Inactivate, x)$, then $\exists\ r_i = (Activate, V_x)$, where $j \geq i$. Moreover, $\forall\ r_i, r_k \in \omega$ such that $r_i = r_k = (Activate, V_x), i < k$, there must exist $r_j = (Inactivate, V_x)$ where $i < j < k$.

In addition to the booted and shutdown states (denoted S_0 and S_{n+1} respectively), we assume there are n additional inactive states (denoted $S_1, S_2, \ldots, S_n$) which can be used to hold idle VMs. Let $V^{S_0}, V^{S_1}, \ldots V^{S_n}, V^{S_{n+1}} \subseteq V$ denote the sets of VMs in states $S_0, S_1, \ldots S_{n+1}$ respectively at a given time instant. We also define $V^A \subseteq V^{S_0}$, as the

set of VMs that are actively serving clients at that instant. Moreover, the soft bounds on the maximum number of VMs in states $S_0, S_1, \ldots S_{n+1}$ are denoted as $B_0, B_1, \ldots B_{n+1}$ respectively. Therefore, $\forall i \in \{0, 1, \ldots, n+1\}, |V^{S_i}| \leq B_i$, and $\sum_{i=0}^{n+1} |V^{S_i}| = v$.

Given this notation, we define the *VM configuration* of the system at any instant, as the tuple C, where

$$C = (V^A, V^{S_0}, V^{S_1}, \ldots, V^{S_n}, V^{S_{n+1}}).$$

C defines the state in which each VM is at the particular time instant. Let S_C be the set

$$\{C | \forall i \in \{0, 1, \ldots n+1\}, |V^{S_i}| \leq B_i, \sum_{i=0}^{n+1} |V^{S_i}| = v\}.$$

Thus S_C denotes the space of all possible VM configurations for the given state hierarchy. Any idle VM management policy π is comprised of the tuple (Q, q_0, g) where,

(i) Q is the set of *control-states* of the policy,

(ii) $q_0 \in Q$ is its initial control state,

(iii) $g : S_C \times Q \times (V \times Type) \rightarrow S_C \times Q$ is the *allocation function*.

The allocation function has the property that the target VM of any request r_i (i.e, $V_{r_i.ID}$) is in $C'.V^A$ whenever $g(C, q, r_i) = (C', q')$ and $r_i.Type = Activate$. Similarly, $V_{r_i.ID}$ is not in $C'.C_A$ whenever $g(C, q, r_i) = (C', q')$ and $r_i.Type = Inactivate$.

Therefore, any VM management policy π can be considered a finite state machine with $S_C \times Q$ control-states, inputs $V \times Type$ (i.e., outgoing transitions), and a transition function g.

The total number of control-states for any policy π are $O(\left\{ {v \atop n+2} \right\} \times |Q|)$, where $\left\{ {X \atop Y} \right\}$ denotes the second stirling number which defines the total number of ways of partitioning a set of X elements into Y subsets. Simplifying further, the total number of control-states are

$$O(\left\{ {v \atop n+2} \right\} \times |Q|) = O((n+2)^v \times |Q|) = O(n^v.|Q|).$$

Given α control-states and β inputs, the total number of finite state machines that can be defined is $\alpha.(\alpha^\beta)^\alpha$ or $O(\alpha^{\alpha.\beta})$. This is because each input for each control-state can cause a transition to one of the α states, and each state can be chosen as the initial control-state.

Therefore the total number of possible policies given a control-state space of $O(n^v.|Q|)$, and $2 \times v$ inputs are $O((n^v)^{n^v})$. That is, the size of the policy space is exponential in both the number of states and the number of VMs multiplexed on a machine (with the exponent itself comprising of n and v terms). We believe that it is due to this large policy space that finding the optimal policy is relatively difficult.

5.7 Discussion

Other Virtualization Solutions

Using LXC as our example virtualization solution, we demonstrate that idle VMs can be multiplexed across inactive states. However, our model formulation and lower bound on demand-based policies (in Section 5.2) is applicable to any virtualization solution. Similarly, our experimental analysis to determine the model parameters (in Section 5.3) can also be adapted to any virtualization solution. Our simulator can then be re-used (after encoding the new cost-model), to observe the effect of policies on any VM hierarchy. We defer such extension of this work to other virtualization solutions (e.g. Xen and KVM) to future work. Nevertheless, we believe our evaluation of VM density and miss penalties can benefit existing projects [9,77] which use LXC.

Extending Policies

We have implemented only a few sample, reactive and proactive policies (i.e., LRU, Cascaded Belady's, SlidingWindow, and SlidingWindow+ARMA), using only the personal VEE workload. The goal of this work was to uncover a new methodology to increase VEE density by multiplexing VEEs across states, understand and compare the behaviour of reactive and proactive policies for idle VM management, and to compare reactive policies with our theoretical lower bound. With our example personal VEE workloads, we provide valuable insights into the behaviour of a few policies. However, we defer the extension of this work to a broader range of policies and workloads to future work. Our modular simulator design, which isolates policies, workloads, cost-models (described in Section 5.4.1), ensures that a broader analysis can be easily conducted.

We have provided only one online implementation of the SlidingWindow policy (using the ARMA model). We chose the ARMA predictor since it delivered relatively small prediction error for the current workload. To use the SlidingWindow policy on other

workloads other prediction methods may need to be considered, while demand-based policies such as LRU can be applied directly.

Stochastic Value Analysis of Transition Times

We model the problem by using a mean value analysis of transition times. Hence we derive the model parameters by defining an operating range of state capacities within which transition times are largely constant. We also assume that different VM state transitions can take place in parallel and that transition times remain unchanged. Stochastic analysis can be leveraged to model any variations in transition times. However, additional experimental analysis of density and transition times will be required to obtain parameters for a stochastic value model, which we defer to future work. Additionally, our current experimental analysis of LXC (Section 5.3) identifies a number of barriers to state capacities in LXC, which serve as venues for improving future LXC implementations.

5.8 Chapter Summary

Our proposed personal VEE architecture for data-driven applications requires provisioning one VEE per user. VM density of traditional consolidation methods is bounded by the maximum number of VMs that can be co-hosted on a single machine by the underlying virtualization solution. Provisioning a large number of VMs using existing solutions will require un-affordably large amounts of hardware resources. Therefore, we propose a new methodology that increases VM density for workloads that have relatively large uncorrelated active times, such as personal VEE workloads.

In this methodology, idle VMs are multiplexed across a range of inactive states, where their resource footprint is reduced, at the cost of a small increase in client latency called the miss penalty. However, due to differences in their design and resource requirements, different inactive states have varying VM activation and deactivation times, and VM capacities. Consequently, miss penalties for idle VMs in different inactive states vary significantly. Thus choosing the appropriate policy for multiplexing idle VMs across inactive states is essential for maximizing the VM density while minimizing the miss penalties. We divide the policy space into reactive and proactive policies. To understand and guide our policy selection, we formulate a mathematical model for the problem. This model is used to derive a lower bound on the miss penalties of reactive policies. We

observe some similarities between this problem and the problems of page replacement and multi-level cache management. We use these similarities to formulate a proactive policy (called SlidingWindow+ARMA) which leverages inter-arrival time prediction to reduce miss penalties.

Our model uses the inactive state capacities and transition times as input parameters. Therefore, we use microbenchmarks to measure these values to obtain the model parameters for LXC [22], a widely used OS-level virtualization solution. We then categorize typical personal VEE workloads and use the measured parameters to evaluate different reactive and proactive policies. Our policy comparison and simulations provide a quantification of the miss penalties of different policies and allow us to draw insights into their behaviour.

Chapter 6

Towards Tussle Based Operating Systems

6.1 Introduction

The goal of the personal VEE architecture (described in Chapter 3) was to enable an ecosystem of data-driven applications that preserves users' data privacy. The basis of our approach was to prevent application developers from accessing users' data, for example, by sandboxing native VHome applications or transforming data before it is accessed by cloud-based applications.

However, both users and application developers benefit from providing application developers with access to users' data. For instance, application developers can use the data to serve users with advertisements in-exchange for lower application costs. Access to data would allow application developers to improve their data processing algorithms, leading to improved utility to users. As a specific example, data can be used to improve occupancy prediction for home thermostat control applications. On the other hand, as discussed in Chapter 1, data from smart meters, smartphones, and other sensors, can also be misused to reveal private information about the user, for example, socio-economic status [51], TV viewing [91], driving [104,111,147], sleep [66,98], and physical activity habits [101,102] and other details [129,136].

Application developers and users are thus in a *tussle* [69] over access to sensor data. Application developers desire to have unlimited sensor data access, while users wish to preserve their data privacy. The VHome framework (Chapter 3) always resolves such

tussles in favour of the user and prevents personal VEE-resident applications from relaying any data back to application developers. In sharp contrast, smartphone operating systems (OSes) such as Android, iOS, or Windows Phone, typically resolve data tussles in favour of application developers and require users to grant all of an application's data requests at installation time. As a result, many users avoid using smartphone applications that require access to privacy-sensitive sensor data [67].

In their seminal work, Clark et al. [69] introduced the notion of tussles in cyberspace. By analogy, *we posit that operating systems need to formally recognize tussles and need to implement mechanisms and policies to resolve them*. Providing stakeholders – users and application developers – with a principled way to express their requirements and ensuring that the framework resolves tussles according to prescribed policies enables easier understanding of system behaviour for both users and applications. Moreover, a tussle-based abstraction benefits users by allowing them to gracefully balance application utility against data privacy and it benefits application developers by allowing for a broader distribution of applications.

In this chapter, we ask: how can we best extend an operating system[1] with a tussle-based abstraction? We take the first steps in answering this question by designing a framework to formalize, detect, and resolve data tussles.

Our work makes the following contributions:

- A tussle abstraction for operating systems.
- An identification of mechanisms and policies that underlie a tussle-based framework.
- An outline of various open problems for instantiating a tussle-based framework and directions for future work.

The remainder of this chapter is organized as follows. We formulate the design goals that a tussle-based framework should achieve in Section 6.2 and present an outline of the proposed architecture and its components in Section 6.3. We then derive the abstractions that allow application developers and users to express their data requirements in

[1] We use operating system (OS) to loosely refer to any software that intermediates applications' access to sensor data and other system resources (Table 6.2). This includes smartphone platforms such as Android, Windows Phone, iOS, home sensing platforms such as HomeOS [74], our personal VEE framework – VHome (Chapter 3, [169]), and other OSes running on embedded devices such as the Raspberry-Pi [31], increasingly deployed in cyber-physical systems.

Sections 6.4.1 and 6.4.2 respectively. We discuss the notion of tussle resolution and the enforcement of such resolutions in Section 6.4.3 and 6.4.4 respectively. We conclude the chapter with a discussion of future directions and an overview of related approaches in Section 6.5.

6.2 Design Goals

Our primary design goal is to extend existing OSes with a tussle framework that:

- Provides the stakeholders, that is, users and application developers, with a high-level abstraction for expressing their resource requirements and constraints.
- Provides a set of mechanisms to detect and resolve tussles and ensures the enforcement of the resolutions.

We believe that an ideal tussle framework should demonstrate the following properties.

Expressiveness

Numerous sensor applications exist today and others are rapidly emerging [34]. Therefore, the framework should allow applications to freely express their sensing requirements. These include, the type of sensors the application requires, the frequency at which access is required, and the level of access. At the same time, the framework should allow the user to define acceptable sensor access by applications. This should be provided through a control interface that is *understandable* to the users, for example, by allowing them to list inferences that can or can not be drawn by an application.

Resolution

After the stakeholders have expressed their sensor access constraints, the framework should: (i) detect conflicting requirements (i.e. tussles), and (ii) resolve tussles by balancing the requirements (with user input, if necessary). The amount of utility the application provides is likely to be limited by the user's data privacy requirements. This implies that application utility should degrade *gracefully* with increasingly strict privacy requirements. Note that this is a significant departure from most existing approaches which typically require a user to accept all of an application's sensor access requests at installation time.

Robustness

The framework should guarantee that any sensor data access made by an application respects the tussle resolution. In addition, since many applications rely on the integrity of sensor readings, the framework should guarantee this integrity. A sample application that requires trusted readings to be effective is an energy billing application [169].

Extensibility

The framework should be able to support new sensor types, new algorithms that can draw more sophisticated inferences using existing sensor data, and new ways in which stakeholders may want to express their requirements.

6.3 Architecture Outline

In light of these design goals, we first identify the key elements of a tussle-based framework, and present an outline of its architecture. In the subsequent sections, we describe existing work and determine the extent to which it can be leveraged to design and implement the different components.

Figure 6.1 provides an overview of the proposed framework aimed at sensor data tussles. Application developers and users are provided with interfaces which they use to express their sensing requirements and data privacy requirements respectively. Application developers express their sensing requirements through timing parameters (labeled (t_s, t_w, t_p)). Users express their requirements using inferences (labeled I_1, I_2, I_3). Sections 6.4.1 and 6.4.2 explain these parameters and interfaces in greater detail and their relation to prior work.

The *resolver* receives the requirements from the user and application and resolves tussles by generating an *accord*. The accord defines the time, level, and scope of sensor access for the application. To formulate an accord, the resolver leverages the *InferenceDB*, a continually-updated cloud-hosted database of existing inference algorithms. Section 6.4.3 describes the resolver, accord, and InferenceDB in detail.

The *enforcer*, a component of the OS, *implements* the accord by ensuring that all applications' sensor accesses respect the accord's restrictions. We discuss the enforcer and its components in Section 6.4.4.

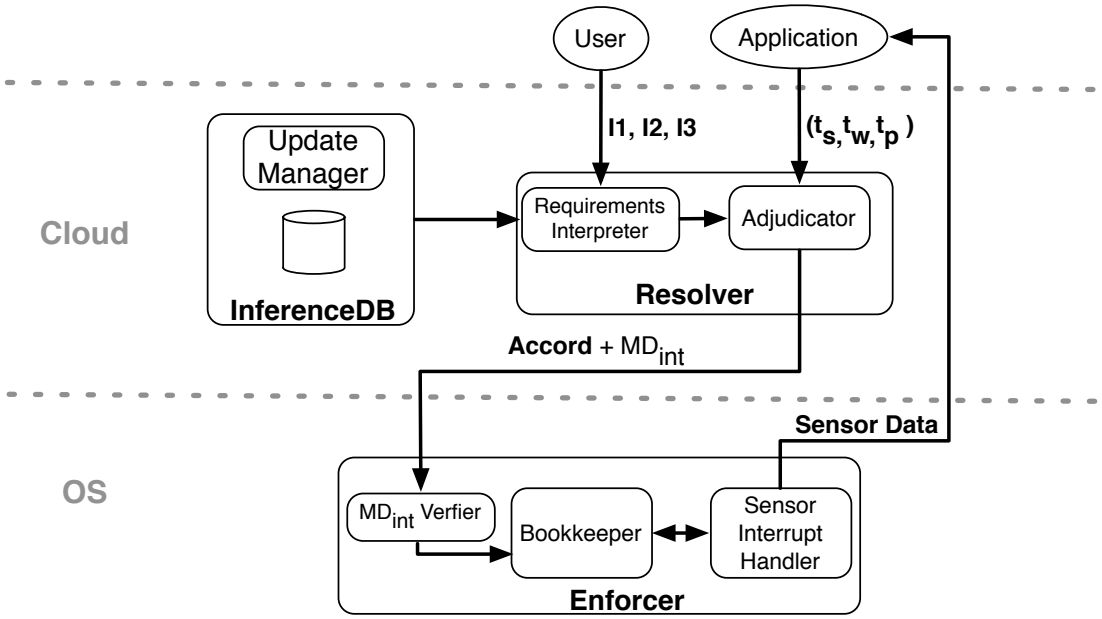


Figure 6.1: Design overview of a tussle-resolving framework.

6.4 Design Challenges

6.4.1 Applications' Sensor Data Requirements

We observe that applications typically collect sensor data, use it to draw inferences about users or the environment, and perform actions based on these inferences. For example, the PreHeat [161] application (described in Section 4.2.1), uses motion sensor data, infers and predicts home occupancy, and adjusts the thermostat accordingly.

Recent work [142] (including ours [170]) has examined methods for allowing applications to request their required *inferences*, through inference-based abstractions, rather than directly receiving actual sensor values (as in the VHome framework; Chapter 3). This enables applications to accurately describe their requirements, while the framework performs the required data processing and delivers the requested inferences. However, this methodology exhibits several deficiencies. Application developers are limited to using inferences supported by the framework. Using a different inference algorithm requires developers to integrate it into the framework (e.g., building an inference-module in Beam [170]). This places additional development and maintenance burdens on the

developer. Moreover, many application developers want their techniques to remain proprietary and may be reluctant to leverage such frameworks. Lastly, the framework may not incorporate certain application-specific design optimizations which the developer wants to employ. Given these drawbacks, we ask: *Is it possible to allow developers to directly express their sensing requirements?*

We survey a range of applications that use sensors across different domains [66, 96, 98, 101, 111, 147, 161]. On analyzing these applications' sensor access, we observe a few similarities, which we explain in the context of a few example applications. We choose these example applications because of the diverse categories they represent (healthcare, safety, and efficiency) and because they process private information about their users. The example applications are:

- **SleepMon** [98, 101]: This application monitors users' sleep quality using their smartphones. It uses the microphone to detect "sleep events", which are used to create a fine-grained sleep profile, compute sleep efficiency, and relate irregular sleep patterns to possible causes.
- **DriverMode** [111, 147]: This application detects when the user is in a moving vehicle, for example, a car, and activates a driver-mode user experience on the user's smartphone. It suppresses non-critical notifications and informs other users. It uses the accelerometer to infer the user's transportation mode and records driving periods for visualization.
- **PreHeat** [161]: Recall that, as described in Chapter 4, this application infers the occupancy of a room or a home. The inferred occupancy is then used to adjust the thermostat to save energy [25, 96, 161].

By analyzing these applications we observe that their sensing requirements, and those of many other applications, can be expressed using the following parameters (illustrated in Figure 6.2).

Sampling rate (t_s)

This parameter defines the minimum time granularity for any batch of sensing values read by the application. For instance, 0.125 ms (or 8 kHz) for the microphone readings used by SleepMon [98, 101], 500 ms (or 2 Hz) for PreHeat's passive infrared (PIR) readings [96], and 100 ms (or 10 Hz) in the case of DriverMode [147]. Application developers

can tune the sampling rate to accommodate their inference algorithms and deployment scenarios. For instance, SleepMon's inference algorithm performs a spectral analysis of the sensor readings using power spectral density. A sampling rate which is too infrequent will prevent the algorithm from being able to differentiate between events (such as snoring, coughing, and moving), thereby impacting its detection of sleep-related events. A sampling rate which is too high results in resource waste, including CPU and battery energy, because the application will need to process a larger volume of readings.

Batch size (t_w)

Most sensing applications require a batch of continuous sensor readings, which they process using inference algorithms. The batch size parameter defines the minimum size of any batch of sensor readings delivered to the application, expressed as a time range. For instance, SleepMon [98] requires sensor readings with t_s=0.125 ms, with a batch size (t_w) of 100 ms, that is, 800 readings sampled at 8 kHz. Similarly, PreHeat [96] has a t_w of 120 seconds. Developers tune the batch size to meet the needs of their inference algorithm. An overly small batch size reduces inference accuracy, for example, lower precision for occupancy, driving, and sleep detection. Moreover, many algorithms require a given minimum batch size in order to detect cyclic patterns or other peculiarities in the data. For instance, inferring if the user is walking using accelerometer readings requires a batch size of at least 3 seconds [112]. In contrast, an overly large batch size decreases algorithm accuracy because multiple events may co-exist within a single batch [147]. It may also lower an algorithm's sensitivity because it increases the number of missed events [147].

Periodicity (t_p)

Most sensing applications perform their batch inference computations periodically (with a period t_p), rather than continuously. For instance, because PreHeat [96] is exclusively interested in inferring occupancy events that last longer than 10 minutes, the occupancy inference computation (processing a 120 seconds batch of PIR readings sampled at 2 Hz) needs to be carried out only once every 10 minutes, meaning t_p = 600 seconds. DriverMode and SleepMon use a t_p value of 5 minutes [98, 111]. Unlike the previous parameters, t_p affects the minimum length of events that are to be detected but it does not impact the underlying inference algorithms' accuracy.

These three parameters allow applications to succinctly express their sensing requirements. Note, however, that applications can still provide reduced functionality as long

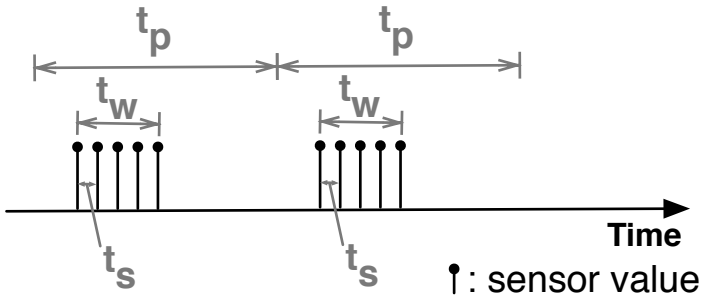


Figure 6.2: Data sampling parameters.

as the parameter values lie within a range of useful values. As explained above, different values within these ranges will impact applications' functioning and utility in different ways. Along the same lines, as we explain below, users' requirements also map to a spectrum. The challenge then lies in determining parameter values that lie within both the stakeholders' operating ranges.

6.4.2 Users' Data Privacy Requirements

Many application frameworks provide users with accept-or-deny sensor controls to express their data privacy requirements. Mobile application platforms such as Android, Windows Phone, and iOS allow the user to grant or deny an application's access to a sensor either at application installation time (in Android and Windows Phone) or when the sensor is first accessed at runtime (in Android M and iOS). Existing work has shown that per-sensor permission models provide very limited insight to the user [82,150,151]. Users have little understanding of which sensors an application should access and why. User-prompts are disruptive, cause "prompt fatigue", and condition users to simply grant all of an application's requests [151].

Other work has focused on alleviating this problem by enabling users to express their data privacy requirements as a set of rules [62,189]. Users specify a set of privacy rules which the OS enforces. Rules can be of the form "block sensor S for a given application" [189], or can be tuples, (C, S, A), where C specifies the context under which a given sensor S is accessed, and A specifies an action (such as perturbation) which needs to be performed on the sensor data before providing it to an application [62]. Although this approach provides greater access control, it burdens the user with the task of updating the rules as the number of sensors and applications increases and with the evolution of

inference algorithms that can be misused to reveal private information.

Chakraborty et al. [62] propose an approach where users are provided with high-level inference abstractions to express their requirements. Users specify two prioritized lists of inferences, comprised of inferences that are acceptable and unacceptable to them respectively. The priority value indicates the importance the user assigns to an inference. An *a priori* mapping of sensors (and their combinations) to inferences (and inference accuracies) that incorporates all known inference algorithms is assumed. Given the two inference lists, the framework generates a list of sensors an application can access. It computes an optimal sensor assignment for applications, maximizing the accuracy of the acceptable inferences, and minimizing the accuracy of the unacceptable inferences. This approach provides users with an understandable, low-overhead, high-level interface, but it suffers from some shortcomings that we discuss in detail in Section 6.5.2.

Inference-based interfaces are intuitive, however, they lack *interpretability*: for instance, an application that infers user occupancy at a time granularity of one minute gains more information about the user than an application with a granularity of one hour, even though both applications use the same inference. Therefore, we envision a user interface where users not only list their acceptable and unacceptable inferences, but, for each acceptable inference, are provided with high-level interpretable controls. One example would allow the user to specify the acceptable time granularity at which applications are able to infer his or her occupancy (such as one hour) and to also control the granularity of occupancy events the application can capture (such as occupancy events longer than 10 minutes).

6.4.3 Tussle Resolution

As described above, users express their requirements using an inference-based control interface, whereas applications express their requirements using a formal description, such as (t_s, t_w, t_p) tuples, for sensing requirements. The *resolver* (shown in Figure 6.1) is a service that processes a given set of applications' sensing requirements and a set of user's data privacy requirements to produce the tussle resolution, referred to as the *accord*.

Since users express their requirements through inferences, the resolver contains a *requirements interpreter* which interfaces with the *InferenceDB*. The requirements interpreter maps the user's requirements to resource-specific parameters such as sensor-specific bounds on parameters t_w and t_p. The *InferenceDB* is a service which provides the requirements interpreter with a mapping of inference types and accuracies to sensor types

and sensing rates. For instance, it would map "Occupancy inferences with 100% accuracy" to the required motion sensor sampling rates. We envision that the InferenceDB would be hosted by trusted third-party providers, for example, as public-facing web services in exchange for user payments (or perhaps, freely from trusted open-source providers). These third parties would be responsible for updating their respective InferenceDBs with existing and newly discovered inference algorithms.

After the user's requirements are converted by the requirements interpreter, they are provided to the *adjudicator*. The *adjudicator* balances the user's and application's requirements and resolves any conflicting requirements using a policy. The policy specifies parameters that map different levels of sensing to different levels of application utility and functionalities. This allows the user to interpret the impact of privacy requirements on application experience.

For some users, the process of fine-tuning privacy requirements may be too complicated or time consuming. A possible solution is to provide users with access to an open-source catalog of trusted, common privacy profiles, from which a user may select a desired set of policies. This is akin to "battery profiles" commonly found on smartphone and tablet OSes. Alternatively, another method could rely on asking users a small number of high-level, privacy-related questions, in order to automatically build a customized privacy profile tailored to the user's needs.

As sensors and inference algorithms evolve, both the requirements interpreter and adjudicator may need to perform significant amounts of computation, as they are essentially solving an optimization problem. Therefore, we envision hosting the resolver in the cloud, either with a trusted third party provider, or on a user's personal VEE (Chapter 3).

Applications present their requirements to the OS at the time of instantiation, that is, at the time of initialization. Similarly, the user presents his or her preferences to the OS via a suitable user interface periodically, or at the time of application installation. Both sets of requirements are then forwarded to the resolver. Figure 6.3 illustrates a sample request sent from the OS to the resolver.

We assume a two-way encrypted channel of communication between the resolver and the OS. The resolver responds with the corresponding accord along with integrity metadata MD_{int}. MD_{int} is computed as follows:

$$MD_{int} = Sig_{K_{priv}^{Resolver}}(H[Request]||H[Accord]).$$

As described in Table 6.1, MD_{int} is a signed hash of the request and the corresponding accord. MD_{int} ensures integrity of the accord, prevents replay attacks, and allows the

```
User: Occupancy - 10 min, Activity - 30 min,
Sleep - 1 hr, ...
App-1: Mic: (8 kHz, 0.125 ms, 1800 s)
App-2: Accelerometer : (10 Hz, 120 s, 300 s)
App-3: PIR: (2 Hz, 60 s, 600 s)
```

Figure 6.3: Example resolver request.

device OS to determine if an accord matches the last issued resolver request. This design allows the resolver to be stateless, ensuring easy scalability for a large number of devices and users, which is neccessary if the resolver is to be hosted as a trusted third party service.

$H[x]$: Cryptographic hash of x
Sig_K[x]: Digital signature of x with the key K
K_{pub}^{owner}, K_{priv}^{owner}: a public-private key pair of $owner$
$\|\|$: Concatenation

Table 6.1: Glossary.

6.4.4 Resolution Enforcement

After an accord has been formulated, the OS needs to ensure that it is obeyed. The *enforcer* (shown in Figure 6.1) is an OS component that ensures that any resource access by any application conforms to the accord generated by the resolver.

Upon receiving an accord from the resolver, the enforcer first verifies if the accord matches the current resolution request issued by the OS (Figure 6.3). It then decodes and stores the accord in memory. To ensure that application resource accesses respect the accord, the enforcer maintains bookkeeping information. For instance, for sensor data tussles, the enforcer records the types, levels, and extents of sensor data accesses by applications.

If the OS runs on a user-device such as a smartphone or PC, the enforcer also handles sensor interrupts: upon receipt of an interrupt, the enforcer consults its bookkeeping information and any accords, and determines the set of applications to which the data can be permissibly delivered, for example, through an asynchronous callback. If the OS

runs on a VEE in the cloud, the enforcer is comprised of only the bookkeeper and verifier modules.

6.5 Discussion

6.5.1 Prior Work

Clark et al. [69] recognize tussles between stakeholders in the networking space and outline solutions to resolve them. Likewise, we propose recognizing tussles among stakeholders on operating systems on commodity devices and outline mechanisms to detect and resolve them.

Existing work has proposed several additions to existing OSes to handle sensors and actuators. Many systems have focused on either detecting overprivileged applications on smartphones [81,185], or tracking the flow of data to applications to detect malicious applications [78]. Spahn et al. [173] design an OS service that discovers application-level data objects, for example, emails and documents, and provides users with unified object management. Santos et al. [156] propose a lease-based allocation of resources on smartphone OSes to enable verifiable application behaviour. These approaches are complementary to ours. They focus on providing mechanisms to address various relevant sub-problems, however, they do not recognize tussles between stakeholders, which are the main cause of these problems. We propose a framework to recognize, detect, and resolve these tussles, which can be further enriched by incorporating these existing mechanisms. Haddadi et al. [95] propose a trusted arbitrator for allowing users to control applications' access to historical data. We provide a framework for stakeholders to express their resource access requirements, that can be used to view historical data as a resource.

Prior work has made several attempts to manage data tussles [46,53,62,124,139,189] but when compared to a tussle-based framework, they suffer from several shortcomings: their allocation of sensing capabilities to applications is unverifiable, they overlook balancing application utility and data privacy, do not guarantee authenticity of sensor data, and require constant device software updates to handle new sensors and inference algorithms.

6.5.2 Open Problems

Resource Tussles

We provide a way for application developers and users to express their sensor data requirements, but these stakeholders often also enter into tussles over access to other system resources. For instance, concurrent applications compete to access computational resources such as CPU time, memory, and network bandwidth. Users often want to exercise some control over resource allocation, to understand the system behaviour, and prioritize different applications [128]. Table 6.2 lists some of the key system resources present today on user devices, including computational resources, sensors, actuators, and even *software resources*, such as user-generated content (e.g., photos and videos), network ports, and file descriptors. Therefore, we require a way to enable applications to express their resource requirements.

Resource	**Examples**
Traditional	CPU, RAM, disk space, disk and network bandwidth
Sensors	Microphone, camera, GPS, accelerometer
Actuators	Electrical switch, thermostat, lock, display
Data	Sensor data streams, user-generated content (contacts, images, etc.)

Table 6.2: System resources.

Applications must also express the durations for which they require access to a resource, and the size of their requirement such as percentage of CPU time or amount of memory. This implies that the interface between the application and the OS (usually the process abstraction) may need to be altered to give the OS a non-black box view into the application's consumption of resources. Many modern smartphone OSes, including Android and Windows Phone, have adopted this approach, with applications divided into UI-intensive components (e.g., Android activities [5]) and computation-intensive components (e.g., Android services [5]). Unfortunately, these abstractions only help to implement specific static policies defined by the OS-provider, such as maximizing battery life. For designing a tussle-based framework, existing unified resource abstractions such as Fence [128] may be leveraged. However, they need to be supplemented with: (i) a language to allow applications to freely express their resource requirements, and (ii) policies that resolve any and all resource tussles. This will allow users to control and

reason about resource tussles in the same manner as data tussles. We envision that for resolving resource tussles, the enforcer would need to be suitably enriched to encompass a CPU scheduler, memory manager, a disk manager, and a network manager.

User-Controlled Data Privacy

As discussed in Section 6.4.2, we require a mechanism that allows users to express their data privacy requirements in an interpretable way, and then maps those requirements to appropriate constraints on sensor access levels. However, the amount of information revealed by sampling a sensor at two different sampling granularities can vary significantly. For instance, an application that accesses the accelerometer to infer if the user is in motion requires a much smaller time granularity than another application that infers the user's physical activity. Similarly, an application accessing GPS at a time granularity of one minute and a space granularity of ten meters can infer significantly different user information than an application with respective time and space granularities of one hour and ten kilometers. As discussed in Section 6.4.2, the ipShield [62] approach provides users with an understandable, low-overhead, high-level interface for managing data privacy but it does not take sensor sampling granularity into consideration. Moreover, ipShield's [62] optimization objective function minimizes the inference accuracy of unacceptable inferences but it does not provide the user with any interpretable metric that conveys the amount of private information revealed. Nevertheless, it serves as a good starting point for building other user-understandable and quantifiable approaches to handle sensor data privacy.

Trusted Readings

Our current work assumes that the OS is trusted and does not tamper with sensor readings. Trusted readings are required in many applications such as billing [169] and geofencing [130]. However, in practice, it may be difficult to guarantee that a particular OS is trusted. Moreover, the user may want to provide readings that are perturbed or otherwise modified, for example to spoof readings during development or replay previous GPS traces to shield their current location [62]. Therefore, perturbations to sensor readings by the user, and the requirement of unperturbed readings by applications can also be viewed as a tussle. A possible solution is to include a noise coefficient with sensor requests, $0 \leq t_n \leq 1$. A coefficient of zero corresponds to unperturbed readings, whereas a coefficient of one could correspond to entirely fabricated readings. The resolver can then

balance an application's requested noise coefficient against the user's privacy requirements, while the enforcer ensures that readings delivered to applications are perturbed to the agreed upon level.

Existing work has shown that trusted readings can be provided despite an untrusted OS by securing appropriate components, such as by using a secure execution mode in modern CPUs [130]. However, this approach [130] requires sensor drivers to be a part of the Trusted Computing Base (TCB), and thus increases its volume and hence its vulnerabilities [157]. A potential solution is to partition sensor drivers' functionality so that only their security-critical components contribute to the TCB.

Tussles on a Cloud-Hosted OS

Applications such as Nest [25] use sensors deployed in users' homes and backhaul data to Nest servers in the cloud which host the application logic. This is problematic in two ways. First, users' sensor data needs to be transferred to the application developer's server, even though data processing can be hosted on users' personal VEE. Second, the cloud server hosting the application logic has no means to allow users to express their data privacy requirements.

What is needed, therefore, is an instantiation of a tussle-based framework on the device such as Nest [25]. This framework may coordinate with instantiations on other user devices and the application developers' cloud servers. The challenge, therefore, lies in designing a tussle-based framework for such IoT devices.

Verifiable OS Implementation

A potential direction for building tussle-oriented operating system frameworks is to use a Unikernel [131]. In this approach, the OS kernel is specialized to support only a pre-specified set of applications. Moreover, the entire kernel is written in a strongly-typed language (OCaml) that allows the OS's correctness to be formally proved, in the sense of always obeying certain high-level assertions. We believe that tussle accords can be formally expressed in terms of these assertions, thus a Unikernel that implements a tussle accord can be trusted to enforce the accord.

6.6 Chapter Summary

We observe that application developers and users often have mutually conflicting objectives with regards to accessing sensor data. Application developers desire free access to sensor data to advance their data processing algorithms and techniques. On the other hand, users are often weary of unwarranted private information being leaked through their sensor data and hence wish to preserve their data privacy. This situation results in a tussle over sensor data access between the two stakeholders – users and application developers. Moreover, such tussles also occur when stakeholders access many other system resources. We advocate that operating systems need to recognize these conflicts and provide suitable mechanisms and policies to detect and resolve them. We outline the design of a framework which: provides applications and users with high-level interfaces to express their sensor data requirements, detects conflicting requirements, resolves them, and ensures the implementation of the resolution. We identity various open problems that need to be solved to extend existing operating systems with a tussle-based framework.

Chapter 7

Conclusion and Future Work

This chapter concludes the dissertation. In Section 7.1, we summarize the main contributions of our work. We outline directions for future work in Section 7.2 and present concluding remarks in Section 7.3.

7.1 Summary and Contributions

Due to a sharp decrease in hardware costs and form factor in recent years, sensors have become ubiquitous. A variety of sensors are being embedded into user devices such as smartphones, tablets, and wearable devices. Our homes, buildings, and automobiles are also being increasingly outfitted with sensors. On one hand, data collected from sensors is used to fuel a variety of applications including those for energy efficiency, safety, and healthcare. On the other hand, sensor data can be processed to reveal unwarranted private information about the user. As a result, many users avoid using such applications that require access to private sensor data to protect their privacy. It is therefore imperative to preserve data privacy while enabling a rich ecosystem of applications that process users' private sensor data.

The focus of this dissertation has been on building a system that enables data-driven applications while preserving data privacy, without burdening users with the tasks of provisioning durable data storage and computational resources for the applications. Our proposed approach leverages users' personal virtual execution environments (VEEs) hosted in the cloud to create an ecosystem of privacy-preserving applications of personal

data. We examine the feasibility of the proposed architecture by providing a proof-of-concept instantiation targeting home energy as an example use case. We observe that applications have some specific data storage and retrieval requirements for sensor data. Consequently, we build a data management system for time series sensor data to meet these requirements while leveraging commodity storage services providers. We also observe that our approach provides each user with a personal VEE and will require provisioning a large numbers of personal VEEs for supporting large numbers of users. Therefore, we develop a new methodology to provision a large number of VEEs with low-duty cycle workloads on a single machine. We formulate the theoretical foundations of the methodology and demonstrate its application to an example virtualization solution.

The key insight underlying our proposed architecture is that allocating appropriate levels of commodity services offered by modern clouds directly to users (rather than application developers) enables users to host both their data and their desired data-driven applications within their confines; thus enabling a large ecosystem of data-driven applications *and* preserving users' data privacy. Our work has to been on designing and implementing the mechanisms, techniques, and methodologies to realize this vision. The key contributions can be summarized as follows.

As justified in Chapter 3, the initial focus of our proposed architecture is home energy data. Our main contributions in this chapter are as follows.

- We present the architecture of a system that allows users to own and control access to their home energy consumption data and freely use data-driven applications of their choice.
- We demonstrate the feasibility of the architecture by building a prototype implementation using commodity cloud computing platforms.
- We present a qualitative evaluation of the prototype implementation with respect to data privacy and data use, and describe how it meets our design goals.

The main architectural components that we design and implement include, (i) an in-home gateway for relaying sensor data and control commands to the personal VEE, (ii) a personal VEE framework which warehouses home energy data, exposes it to third-party applications via APIs, and provides data access control, and (iii) sample home energy applications that run within an instance of VHome. Our personal VEE framework also enables applications to securely and privately control some home appliances.

As described in Chapter 4, existing sensor data storage systems (including our prototype implementation in Chapter 3) do not meet all the data management requirements of applications. Our work in Chapter 4 addresses this problem by designing a storage system for sensor data and makes the following contributions.

- We survey a wide range of data-driven applications and present a formulation of their data management requirements.
- We describe the design and implementation of Bolt, a system for storage, retrieval, and sharing of in-home sensor data that meets the applications' requirements.
- We present a performance evaluation of Bolt using three sample applications. We demonstrate that, when compared with OpenTSB [27], Bolt's use of chunking, segmentation, and index-DataLog separation techniques leads to a decrease in data retrieval time of up to a factor of 40, and a 3 to 5 times reduction in storage space used.

Moreover, Bolt provides applications with programming abstractions for storage, range-query based retrieval, and sharing of sensor data, thus simplifying application development.

As described in Chapter 5, we propose to host large numbers of personal VEEs on machine by multiplexing VMs across multiple inactive states. Our insight is that many workloads executed on personal VEEs would exhibit frequent, often long, and uncorrelated idle periods. Therefore, existing work on inactive states for reducing the resource footprint of idle VMs can be leveraged to maximize the number of VMs hosted on a machine at the cost of a small latency penalty for client requests (called miss penalty). Our work in Chapter 5 explores this design alternative and makes the following key contributions:

- We present a formal model for policies for managing idle VMs across inactive states and derive a lower bound on the miss penalty of reactive policies.
- We present a measurement of model parameters using microbenchmarks with LXC [22] as our example virtualization solution.
- We present a study of a few representative idle VM management policies, quantify their miss penalties, and draw insights into their behaviour.

We design and implement a simulator for studying the behaviour of the idle VM management policies. Our simulator is extensible to other policies, to other types of workloads, and to inactive state hierarchies of other virtualization solutions.

In summary, the architecture, techniques, mechanisms, and methodologies proposed in this dissertation present a viable solution to achieve the seemingly conflicting requirements of enabling *data privacy* and *data use*.

7.2 Future Work

In addition to the topic-specific future work directions presented in Sections 3.5, 4.7, and 5.7, we present the following avenues for future work.

7.2.1 Tussle Framework for IoT

In Chapter 6, we outline a tussle framework for a single device. However, in many pervasive sensing environments, sensor data from different devices can not only be correlated to draw inferences about a given user but also about a larger number of users, for example, co-workers in an office. Moreover, the data privacy requirements and the acceptable loss of data privacy across different users may vary significantly. What is needed, therefore, is a tussle framework that allows the stakeholders to express their data privacy requirements and their acceptable application utility in this broader context. The framework then bears the onus for interfacing with sensors embedded across multiple devices and physical spaces, as well as detecting and resolving the tussles in keeping with the stakeholders' specified requirements.

7.2.2 Virtualization for High Density Hosting

In Chapter 5, we formulate the theoretical foundations for state-based multiplexing of low duty-cycle VMs and study the behaviour of different policies. An open design challenge lies in building a practical low-overhead virtualization solution that not only implements multiple inactive states but also supports easy extension for incorporating idle VM management policies. The behaviour of different policies can then be observed in practice and compared with the expected behaviour (derived in Chapter 5) to identify any discrepancies and performance bottlenecks. Similarly, suitable mechanisms need

to be designed to handle cases where the assumed low duty-cycles of personal VEEs co-hosted on a single machine are violated. We believe that in such scenarios where personal VEEs hosted on a given machine become active together, load balancing can be used to migrate VEEs to other under-utilized machines, for example, those that host personal VEEs for users from a different time zone. Alternatively, personal VEEs can be co-hosted on a single machine while ensuring adequate diversity in their active and idle times. Therefore, suitable analysis methodologies and supplementary mechanisms will need to be designed.

7.2.3 Storage Cost Optimization for Time Series Data

Our storage system in Chapter 4 provides policy-driven storage across providers, that is, allowing applications to define which parts of a given stream of data should be stored with a given provider such as local disk, Windows Azure [37], or Amazon S3 [4]. However, different storage providers have varying pricing schemes depending on the location of the datacenter used and the type of underlying storage medium used (e.g., disk versus solid state drives). Moreover, some applications may require the retrieval of certain parts of data more frequently than other parts. For example, Preheat [161] prioritizes using a recent window of occupancy data over older data, while DNW [56] may issue repeated retrievals of a given time segment where the majority of suspicious activity was recorded. In such cases, it is cumbersome for the application developer to optimize storage costs given the variation in prices and retrievals. However, it may be possible to enrich the different retrieval queries (such as those provided by Bolt in Chapter 4) with cost optimization models. Applications can then specify if they prioritize retrieval time over storage cost (or vice versa). The system would bear the onus for computing the optimal storage strategy and updating the cost models accordingly.

7.2.4 Control Architecture for IoT

Many data-driven applications make use of actuators such as in-home switches, locks, and thermostats, while many (or all) components of the applications may be hosted in the cloud. In such scenarios, it is imperative to ensure that appliance control remains within certain user-defined safety limits even in the presence of faults. Example faults include home network outages, misconfiguration or malfunction of the sensing or actuating devices, and user overrides. What is needed, therefore, is a cascading control

architecture, where an application's control logic is partitioned into a lightweight fail-safe component and a core data-based component. We believe that, similar to our data framework (described in Chapters 3 and 6), it is possible to design a control framework that interfaces with data-driven applications to facilitate and regulate device actuations.

7.2.5 Semantic Isolation of Applications

Pervasive sensing in homes can give rise to many complexities leading to unintended consequences. This problem is exacerbated by data-driven applications that use sensor data to trigger actuations and alerts. For instance, consider an application that uses home energy consumption to deduce occupancy [115], which is then used to control other appliances or trigger user alerts. In such scenarios, a malfunctioning energy sensor can cause erroneous occupancy inferences, cascading into erroneous appliance actuations of false user alerts. Similarly, in another scenario, consider two applications, one which uses an IR sensor to trigger room lighting and another which infers occupancy from lighting to control home heating (in turn affecting IR levels), thus affecting each other's operations. The two applications thus do not operate in complete semantic isolation. What is needed, therefore, is a mechanism that can detect such conflicts among applications (possibly at installation time) and can alert the user and hence prevent such fallacious consequences.

7.3 Concluding Remarks

Today most data-driven applications are being built in a service provider-centric way where users are required to transfer their data to the application service providers in order to use the provided applications. We seek to ease this situation by proposing our personal VEE approach where application executables are transferred to user controlled environments and run within the users' confines. This enables an ecosystem of privacy-preserving data-driven applications, and unlike the service provider based approaches, does not lead to private data of thousands of users being stored under one administrative domain where it may susceptible to large scale breaches. This work addresses the design of mechanisms, techniques, and methodologies to overcome the barriers to the feasibility and adoption of the personal VEE approach by large numbers of users.

The ongoing evolution of new sensors, sensing capabilities, and data-processing algorithms requires extending software systems to provide users with data privacy and

control. As a starting point, our work has focused on in-home sensors. However, we believe all systems that collect, archive, and process data concerning users need to recognize users as stakeholders and need to allow users to freely choose their data control policies. This includes sensor systems in office and commercial buildings, public spaces, automobiles, public transportation, and in other environments likely to be outfitted with sensors in the future. The insights we draw through our work are equally applicable to such other domains.

References

[1] 90 Million Homes Worldwide will Employ Home Automation Systems. https://www.abiresearch.com/press/90-million-homes-worldwide-will-employ-home-automa.

[2] Aeon Labs Smart Energy Switch. http://www.aeon-labs.com.

[3] Amazon Home Automation Store. http://www.amazon.com/home-automation-smarthome/b?ie=UTF8&node=6563140011.

[4] Amazon Web Services (AWS). http://aws.amazon.com.

[5] Android SDK. http://developer.android.com.

[6] Apache HBase. http://hbase.apache.org.

[7] BizEE Energy Lens. http://www.energylens.com.

[8] Current Cost Envi CC-128. http://www.currentcost.com.

[9] Docker: An Open Platform for Distributed Applications for Developers and Sysadmins. http://www.docker.com.

[10] Dropcam - Super Simple Video Monitoring and Security. https://www.dropcam.com.

[11] Facebook Surpasses One Billion Users. http://www.bbc.com/news/technology-19816709.

[12] Fast, Portable, Binary Serialization for .NET. http://code.google.com/p/protobuf-net.

[13] FitBit Third-party Apps. http://www.fitbit.com/apps.

[14] FitBit Ultra. http://www.fitbit.com/product.

[15] Google Powermeter. http://www.google.com/powermeter.

[16] Green Button Connect. http://www.greenbuttonconnect.com.

[17] Green Button Initiative. http://www.greenbuttondata.org.

[18] Home Energy Saver. http://hes.lbl.gov/consumer.

[19] Home Energy Yardstick. http://www.energystart.gov.

[20] IFTTT: Put the Internet To Work for You. https://ifttt.com.

[21] JouleBug. http://www.joulebug.com.

[22] Linux Containers. http://lxc.sourceforge.net.

[23] Linux VServer. http://linux-vserver.org.

[24] Microsoft Hohm. http://www.microsoft-hohm.com.

[25] Nest. http://www.nest.com.

[26] Ontario Time-Of-Use Pricing. http://www.ontario-hydro.com.

[27] OpenTSDB. http://www.opentsdb.net/.

[28] OpenVZ. http://openvz.org.

[29] Pachube-Cosm Ltd. http://www.cosm.com.

[30] Philips Hue. http://www.meethue.com.

[31] Raspberry Pi. https://www.raspberrypi.org.

[32] San Diego Gas and Electric. http://www.sdge.com.

[33] Sungg Pro Energy Audit Tool. http://www.snuggpro.com.

[34] The Internet of Things. http://share.cisco.com/internet-of-things.html.

[35] The Locker Project. http://lockerproject.org.

[36] Waterloo North Hydro Corp. http://www.wnhwebpresentment.com/app.

[37] Windows Azure. http://azure.microsoft.com.

[38] Withthings Smart Devices. http://www.withings.com.

[39] Z-Wave Alliance. http://www.z-wavealliance.org.

[40] The US Market for Home Automation and Security Technologies. Technical report, BCC Research, 2011.

[41] Monitoring Report Smart Meter Deployment and TOU Pricing. Technical report, Ontario Energy Board, 2012.

[42] Utility-Scale Smart Meter Deployments, Plans & Proposals. Technical report, Institute for Electric Efficiency (IEE), Edison Foundation, 2012.

[43] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of CIDR*, 2005.

[44] Gergely Ács and Claude Castelluccia. I Have a DREAM!: Differentially Private Smart Metering. In *Proc. of International Conference on Information Hiding*, 2011.

[45] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *ACM SIGOPS Operating Systems Review*, 2002.

[46] Yuvraj Agarwal and Malcolm Hall. ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing. In *Proc. of ACM MobiSys*, 2013.

[47] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. Principles of Optimal Page Replacement. *Journal of the ACM (JACM)*, 1971.

[48] Rui Araújo, A Igreja, Ricardo de Castro, and Rui Esteves Araujo. Driving Coach: A Smartphone Application To Evaluate Driving Efficient Patterns. In *Proc. of IEEE Intelligent Vehicles Symposium (IV)*, 2012.

[49] Martin F Arlitt and Carey L Williamson. Web Server Workload Characterization: The Search for Invariants. *ACM SIGMETRICS Performance Evaluation Review*, 1996.

[50] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proc. of SOSP*, 2003.

[51] Christian Beckel, Leyna Sadamori, and Silvia Santini. Automatic Socio-economic Classification of Households Using Electricity Consumption Data. In *Proc. of ACM e-Energy*, 2013.

[52] Laszlo A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*.

[53] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proc. of ACM HotMobile*, 2011.

[54] Benjamin J. Birt, Guy R. Newsham, Ian Beausoleil-Morrison, Marianne M. Armstrong, Neil Saldanha, and Ian H. Rowlands. Disaggregating Categories of Electrical Energy End-use from Whole-house Hourly Data. *Energy and Buildings*, 2012.

[55] Maged NK Boulos, Steve Wheeler, Carlos Tavares, and Ray Jones. How Smartphones are Changing the Face of Mobile and Participatory Healthcare: An Overview. *Biomedical Engineering Online*, 2011.

[56] AJ Brush, Jaeyeon Jung, Ratul Mahajan, and Frank Martinez. Digital Neighborhood Watch: Investigating the Sharing of Camera Data Amongst Neighbors. In *Proc. of ACM CSCW*, 2013.

[57] B. Burke. *RESTful Java with Jax-RS*. O'Reilly Media, 2009.

[58] Ramón Cáceres, Landon Cox, Harold Lim, Amre Shakimov, and Alexander Varshavsky. Virtual Individual Servers As Privacy-preserving Proxies for Mobile Devices. In *Proc. of ACM MobiHeld, 2009*.

[59] Kelly E. Caine et al. DigiSwitch: A Device To Allow Older Adults To Monitor and Direct the Collection and Transmission of Health Information Collected at Home. *Journal of Medical Systems*, 2011.

[60] Fernando L. Camargos, Gabriel Girard, and Benoit D. Ligneris. Virtualization of Linux Servers: A Comparative Study. In *Proc. of Linux Symsposium*, 2008.

[61] Don Carney, U□ ur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proc. of VLDB*, 2002.

[62] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. ipShield: A Framework for Enforcing Context-aware Privacy. In *Proc. of USENIX NSDI*, 2014.

[63] V. Chaudhary, Minsuk Cha, J.P. Walters, S. Guercio, and S. Gallo. A Comparison of Virtualization Technologies for HPC. In *Proc. of IEEE AINA*, 2008.

[64] Jianhua Che, Congcong Shi, Yong Yu, and Weimin Lin. A Synthetical Performance Evaluation of OpenVZ. In *Proc. of IEEE APSCC 2010*.

[65] Dong Chen, Sean Barker, Adarsh Subbaswamy, David Irwin, and Prashant Shenoy. Non-Intrusive Occupancy Monitoring Using Smart Meters. In *Proc. of ACM BuildSys*, 2013.

[66] Zhenyu Chen, Mu Lin, Fanglin Chen, Nicholas D Lane, Giuseppe Cardone, Rui Wang, Tianxing Li, Yiqiang Chen, Tanzeem Choudhury, and Andrew T Campbell. Unobtrusive Sleep Monitoring Using Smartphones. In *Proc. of IEEE PervasiveHealth*, 2013.

[67] Erika Chin, Adrienne Porter Felt, Vyas Sekar, and David Wagner. Measuring User Confidence in Smartphone Security and Privacy. In *Proc. ACM SOUPS '12*.

[68] Chun-Te Chu, Jaeyeon Jung, Zicheng Liu, and Ratul Mahajan. sTrack: Secure Tracking in Community Surveillance. Technical report, Microsoft Research, 2014.

[69] David D Clark, John Wroclawski, Karen R Sollins, and Robert Braden. Tussle in Cyberspace: Defining Tomorrow's Internet. *ACM SIGCOMM CCR*, 2002.

[70] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973.

[71] Antonio Corradi, Mario Fanelli, and Luca Foschini. VM Consolidation: A Real Case Based on OpenStack Cloud. *Future Generation Computer Systems*, 2014.

[72] M. Davis, M. Alexander, and M. Duvall. Total Cost of Ownership Model for Current Plug-in Electric Vehicles. Technical report, Electric Power Research Institue (EPRI), 2013.

[73] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. sMAP: A Simple Measurement and Actuation Profile for Physical Information. In *Proc. of ACM SenSys*, 2010.

[74] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An Operating System for the Home. In *Proc. of USENIX NSDI*, 2012.

[75] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Victor Bahl. The Home Needs an Operating System (and an App Store). In *Proc. of ACM HotNets*, 2010.

[76] Carl Ellis, James Scott, Mike Hazas, and John Krumm. Earlyoff: Using House Cooling Rates To Save Energy. In *Proc. of ACM BuildSys*, 2012.

[77] Chris Elsmore, Anil Madhavapeddy, Ian Leslie, and Amir Chaudhry. Confidential Carbon Commuting. In *Proc. of the First Workshop on Measurement, Privacy, and Mobility*, 2012.

[78] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: an Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM TOCS*, 2014.

[79] V.L. Erickson, M.A. Carreira-Perpinan, and A.E. Cerpa. OBSERVE: Occupancy-based System for Efficient Reduction of HVAC Energy. 2011.

[80] Ariel J Feldman, William P Zeller, Michael J Freedman, and Edward W Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *Proc. of USENIX OSDI*, 2010.

[81] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proc. of ACM CCS*, 2011.

[82] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention. In *Proc. of ACM SOUPS*, 2012.

[83] S. Firth, K. Lomas, A. Wright, and R. Wall. Identifying Trends in the Use of Domestic Appliances from Household Electricity Consumption Measurements. *Energy and Buildings*, 2008.

[84] Kevin Fu, Seny Kamara, and Tadayoshi Kohno. Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage. In *Proc. of Networks and Distributed Systems Symposium*, 2006.

[85] Kevin E Fu. *Integrity and Access Control in Untrusted Content Distribution Networks*. PhD thesis, Massachusetts Institute of Technology, 2005.

[86] Flavio D. Garcia and Bart Jacobs. Privacy-friendly Energy-metering Via Homomorphic Encryption. In *Proc. of International Conference on Security and Trust Management*, 2010.

[87] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy. HomeViews: Peer-to-Peer Middleware for Personal Data Sharing Applications. In *Proc. of ACM SIGMOD*, 2007.

[88] Binny S Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions are Better Than Demotions. In *Proc. of USENIX FAST*, 2008.

[89] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proc. of Networks and Distributed Systems Symposium*, 2003.

[90] S. Goyal and J. Carter. A Lightweight Secure Cyber Foraging Infrastructure for Resource-Constrained Devices. In *Proc. of IEEE WMCSA, 2004*.

[91] Ulrich Greveler, Benjamin Justus, and Dennis Loehr. Multimedia Content Identification Through Smart Meter Power Usage Profiles. In *Proc. of CPDP*, 2012.

[92] Eric Griffis, Jeffrey A. Vaughn, and Todd Millstein. A Platform for Expressive and Secure Data Sharing with Untrusted Third Parties. Technical report, UCLA, 2011.

[93] Trinabh Gupta, Rayman Preet Singh, Amar Phanishayee, Jaeyeon Jung, and Ratul Mahajan. Bolt: A Storage System for Connected Homes. In *Proc. of USENIX NSDI*, 2014.

[94] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. Just-in-time Provisioning for Cyber Foraging. In *Proc. of ACM MobiSys 2013*.

[95] Hamed Haddadi, Heidi Howard, Amir Chaudhry, Jon Crowcroft, Anil Madhavapeddy, and Richard Mortier. Personal Data: Thinking Inside the Box. *arXiv:1501.04737*, 2015.

[96] Ebenezer Hailemariam, Rhys Goldstein, Ramtin Attar, and Azam Khan. Real-time Occupancy Detection Using Decision Trees with Multiple Sensor Types. In *Proc. of Symposium on Simulation for Architecture and Urban Design*, 2011.

[97] E. Hammer-Lahav, D. Recordon, and D. Hardt. The OAuth 2.0 Authorization Protocol. *draft-ietf-oauth-v2-18*, 8, 2011.

[98] Tian Hao, Guoliang Xing, and Gang Zhou. iSleep: Unobtrusive Sleep Quality Monitoring Using Smartphones. In *Proc. of ACM SenSys*, 2013.

[99] Mareca Hatler, Darryl Gurganious, and Charlie Chi. Mobile Sensing Health & Wellness. Technical report, ON World, 2013.

[100] Jennia Hizver and Tzi-cker Chiueh. Real-time Deep Virtual Machine Introspection and Its Applications. In *Proc. of ACM VEE*, 2014.

[101] Tâm Huynh and Bernt Schiele. Analyzing Features for Activity Recognition. In *Proceedings of Joint Conference on Smart Objects and Ambient Intelligence: Innovative Context-Aware Services: Usages and Technologies*, 2005.

[102] OzlemDurmaz Incel, Mustafa Kose, and Cem Ersoy. A Review and Taxonomy of Activity Recognition on Mobile Phones. *BioNanoScience*, 2013.

[103] Brendan Jennings and Rolf Stadler. Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management*, 2014.

[104] Derick A Johnson and Mohan M Trivedi. Driving Style Recognition Using a Smartphone As a Sensor Platform. In *IEEE Conference on Intelligent Transportation Systems (ITSC)*, 2011.

[105] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proc. of USENIX FAST*, 2003.

[106] Jerry Kang, Katie Shilton, Deborah Estrin, and Jeff Burke. Self-surveillance Privacy. *Iowa L. Rev.*, 97:809, 2011.

[107] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. A Data Capsule Framework For Web Services: Providing Flexible Data Access Control To Users. *CoRR*, abs/1002.0298, 2010.

[108] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. Secure Data Preservers for Web Services. In *Proc. of USENIX WebApps*, 2011.

[109] S. Kannan, A. Gavrilovska, and K. Schwan. Cloud4Home – Enhancing Data Services with @Home Clouds. In *Proc. of ICDCS, 2011*.

[110] Sudarsun Kannan, Karishma Babu, Ada Gavrilovska, and Karsten Schwan. Vstore++: Virtual Storage Services for Mobile Devices. In *Mobile Computing, Applications, and Services*. 2012.

[111] Aman Kansal, Scott Saponas, AJ Brush, Kathryn S McKinley, Todd Mytkowicz, and Ryder Ziola. The Latency, Accuracy, and Battery (LAB) Abstraction: Programmer Productivity and Energy Efficiency for Continuous Mobile Context Sensing. In *Proc. of ACM OOPSLA*, 2013.

[112] D.M. Karantonis, M.R. Narayanan, M. Mathie, N.H. Lovell, and B.G. Celler. Implementation of a Real-Time Human Movement Classifier Using a Triaxial Accelerometer for Ambulatory Monitoring. *IEEE Transactions on Information Technology in Biomedicine*, 2006.

[113] Jack Kelly and William Knottenbelt. Disaggregating Multi-State Appliances From Smart Meter Data. In *Proc. of ACM SIGMETRICS (adjunct)*, 2012.

[114] Younghun Kim, E.C.-H. Ngai, and M.B. Srivastava. Cooperative State Estimation for Preserving Privacy of User Behaviors in Smart Grid. In *Proc. of IEEE SmartGridComm*, 2011.

[115] Wilhelm Kleiminger, Christian Beckel, Thorsten Staake, and Silvia Santini. Occupancy Detection From Electricity Consumption Data. In *Proc. of ACM BuildSys*, 2013.

[116] Wilhelm Kleimingera, Friedemann Matterna, and Silvia Santinib. Predicting Household Occupancy for Smart Heating Control: A Comparative Performance Analysis of State-of-the-art Approaches. 2013.

[117] Thomas Knauth and Christof Fetzer. Fast Virtual Machine Resume for Agile Cloud Services. In *Proc. of IEEE ICCGC*, 2013.

[118] Thomas Knauth and Christof Fetzer. DreamServer: Truly On-Demand Cloud Services. In *Proc. of SYSTOR*, 2014.

[119] J. Zico Kolter, Siddharth Batra, and Andrew Ng. Energy Disaggregation Via Discriminative Sparse Coding. In *Proc. of NIPS*, 2010.

[120] Janelle LaMarche, Katherine Cheney, Sheila Christian, and Kurt Roth. Home Energy Management Products & Trends.

[121] Nicholas D Lane, Emiliano Miluzzo, Hong Lu, Daniel Peebles, Tanzeem Choudhury, and Andrew T Campbell. A Survey of Mobile Phone Sensing. *IEEE Communications Magazine*, 2010.

[122] Nicholas D Lane, Mashfiqui Mohammod, Mu Lin, Xiaochao Yang, Hong Lu, Shahid Ali, Afsaneh Doryab, Ethan Berke, Tanzeem Choudhury, and Andrew Campbell. BeWell: A Smartphone Application To Monitor. In *Proc. of International ICST Conference on Pervasive Computing Technologies for Healthcare*, 2011.

[123] Sangmin Lee, Rina Panigrahy, Vijayan Prabhakaran, Venugopalan Ramasubramanian, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Validating Heuristics for Virtual Machines Consolidation. Technical report, 2011.

[124] Sangmin Lee, Edmund L. Wong, Deepak Goel, Mike Dahlin, and Vitaly Shmatikov. πBox: A Platform for Privacy-preserving Apps. In *Proc. of USENIX NSDI*, 2013.

[125] Fengjun Li, Bo Luo, and Peng Liu. Secure Information Aggregation for Smart Grids Using Homomorphic Encryption. In *Proc. of SmartGridComm*, 2010.

[126] Jinyuan Li, Maxwell N Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proc. of USENIX OSDI*, 2004.

[127] Ming Li, Shucheng Yu, Kui Ren, and Wenjing Lou. Securing Personal Health Records in Cloud Computing: Patient-centric and Fine-grained Data Access Control in Multi-owner Settings. In *Security and Privacy in Communication Networks*. 2010.

[128] Tao Li, Albert Rafetseder, Rodrigo Fonseca, and Justin Cappos. Fence: Protecting Device Availability with Uniform Resource Control. In *Proc. of USENIX ATC*, 2015.

[129] M.A. Lisovich, D.K. Mulligan, and S.B. Wicker. Inferring Personal Information from Demand-Response Systems. *IEEE Security Privacy*, 2010.

[130] He Liu, Stefan Saroiu, Alec Wolman, and Himanshu Raj. Software Abstractions for Trusted Sensors. In *Proc. ACM MobiSys*, 2012.

[131] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *ACM SIGPLAN Notices*, 2013.

[132] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud Storage with Minimal Trust. *TOCS*, 2011.

[133] Paul B. Menage. Adding Generic Process Containers To the Linux Kernel. In *Proc. of Linux Symposium*, 2007.

[134] Terence C. Mills. *Time Series Techniques for Economists*. Cambridge University Press, 1991.

[135] Emiliano Miluzzo, Ramon Caceres, and Yih-Farn Chen. Vision: mClouds Computing on Clouds of Mobile Devices. *Proc. of International Workshop on Mobile Computing and Services*, 2012.

[136] Andrés Molina-Markham, Prashant Shenoy, Kevin Fu, Emmanuel Cecchet, and David Irwin. Private Memoirs of a Smart Meter. In *Proc. of ACM BuildSys*, 2010.

[137] Richard Mortier, Chris Greenhalgh, Derek McAuley, Alexa Spence, Anil Madhavapeddy, Jon Crowcroft, and Steve Hand. The Personal Container. *Digital Futures*, 2010.

[138] Min Mun, Shuai Hao, Nilesh Mishra, Katie Shilton, Jeff Burke, Deborah Estrin, Mark Hansen, and Ramesh Govindan. Personal Data Vaults: A Locus of Control for Personal Data Streams. In *Proc. of ACM CoNext, 2010*.

[139] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proc. of ACM ASIA CCS*, 2010.

[140] Dennis J Nelson. Residential Baseload Energy Use: Concept and Potential for AMI Customers. In *ACEEE Summer Study on Energy Efficiency in Buildings*, 2008.

[141] Ben Newton, Kevin Jeffay, and Jay Aikat. The Continued Evolution of Web Traffic. In *Proc. of IEEE MASCOTS*, 2013.

[142] Shahriar Nirjon, Robert F. Dickerson, Philip Asare, Qiang Li, Dezhi Hong, John A. Stankovic, Pan Hu, Guobin Shen, and Xiaofan Jiang. Auditeur: A Mobile-cloud Service Platform for Acoustic Event Detection on Smartphones. In *Proc. of ACM MobiSys*, 2013.

[143] Pradeep Padala, Xiaoyun Zhu, Zhikui Wang, Sharad Singhal, and Kang G. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. Technical report, HP, 2007.

[144] Raluca Ada Popa, Jacob R Lorch, David Molnar, Helen J Wang, and Li Zhuang. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proc. of USENIX ATC*, 2011.

[145] Md Anindya Prodhan and Kamin Whitehouse. Hot Water DJ: Saving Energy by Pre-mixing Hot Water. In *Proc. of ACM BuildSys*, 2012.

[146] S. Raj Rajagopalan, Lalitha Sankar, Soheil Mohajer, and H. Vincent Poor. Smart Meter Privacy: A Utility-Privacy Framework. *CoRR*, 2011.

[147] Sasank Reddy, Min Mun, Jeff Burke, Deborah Estrin, Mark Hansen, and Mani Srivastava. Using Mobile Phones To Determine Transportation Modes. *ACM Transactions on Sensor Networks (TOSN)*, 2010.

[148] Alfredo Rial and George Danezis. Privacy-preserving Smart Metering. In *Proc. of ACM WPES*, 2011.

[149] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Incorporated, 2007.

[150] Franziska Roesner, Tohru Kohno, Alexander Moshchuk, Bryan Parno, Harry Jiannan Wang, and Crispin Cowan. User-driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proc. of IEEE Symposium on Security and privacy*, 2012.

[151] Franziska Roesner, David Molnar, Alexander Moshchuk, Tadayoshi Kohno, and Helen J Wang. World-driven Access Control for Continuous Sensing. In *Proc. of ACM CCS*, 2014.

[152] Cristina Rottondi, Giacomo Vertical, and Antonio Capone. A Security Framework for Smart Metering with Multiple Data Consumers. In *Proc. of IEEE INFOCOMM Workshop on Green Networking and Smart Grid*, 2012.

[153] Sushmita Ruj, Amiya Nayak, and Ivan Stojmenovic. A Security Architecture for Data Aggregation and Access Control in Smart Grids. *CoRR*, 2011.

[154] Brandon Salmon, Steven W Schlosser, Lorrie F Cranor, and Gregory R Ganger. Perspective: Semantic Data Management for the Home. In *Proc. of USENIX FAST*, 2009.

[155] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. of Summer USENIX conference*, 1985.

[156] Nuno Santos, Nuno O Duarte, Miguel B Costa, and Paulo Ferreira. A Case for Enforcing App-Specific Constraints To Mobile Devices by Using Trust Leases. In *Proc. of USENIX HotOS*, 2015.

[157] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM TrustZone To Build a Trusted Language Runtime for Mobile Applications. In *Proc. of ACM ASPLOS*, 2014.

[158] Mahadev Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 2009.

[159] Eve M. Schooler, Zhang Jianqing, Adedamola Omotosho, Jessica McCarthy, Zhao Meiyuan, and Li Qinghua. The Trusted Personal Energy Cloud for the Smart Home. *Intel Technology Journal*, 2012.

[160] Gideon Schwarz. Estimating the Dimension of a Model. *The Annals of Statistics*, 1978.

[161] J. Scott, A.J. Brush, J. Krumm, B. Meyers, M. Hazas, S. Hodges, and N. Villar. PreHeat:Controlling Home Heating Using Occupancy Prediction. In *Proc. of ACM UbiComp*, 2011.

[162] Ilari Shafer, Raja R. Sambasivan, Anthony Rowe, and Gregory R. Ganger. Specialized Storage for Big Time Series. In *Proc. of USENIX HotStorage*, 2013.

[163] A. Shakimov, H. Lim, R. Caceres, L.P. Cox, K. Li, Dongtao Liu, and A. Varshavsky. Vis-a-Vis: Privacy-preserving Online Social Networking via Virtual Individual Servers. In *Proc. of COMSNETS, 2011*.

[164] Amre Shakimov, Alexander Varshavsky, Landon P. Cox, and Ramón Cáceres. Privacy, Cost, and Availability Tradeoffs in Decentralized OSNs. In *Proc. of ACM WOSN, 2009*.

[165] Elaine Shi, T.-H. Hubert Chan, Eleanor G. Rieffel, Richard Chow, and Dawn Song. Privacy-Preserving Aggregation of Time-Series Data. In *Proc. of Networks and Distributed Systems Symposium*, 2011.

[166] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for Untrusted Cloud Storage. In *Proc. of ACM CCSW*, 2010.

[167] Rayman Preet Singh, Tim Brecht, and S Keshav. Towards VM Consolidation Using a Hierarchy of Idle States. In *Proc. of ACM VEE*, 2015.

[168] Rayman Preet Singh, Tim Brecht, and Srinivasan Keshav. IP Address Multiplexing for VEEs. *ACM SIGCOMM CCR*, 2014.

[169] Rayman Preet Singh, S. Keshav, and Tim Brecht. A Cloud-based Consumer-centric Architecture for Energy Data Analytics. In *Proc. of ACM e-Energy, 2013*.

[170] Rayman Preet Singh, Chenguang Shen, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. A Case for Ending Monolithic Apps for Connected Devices. In *Proc. of USENIX HotOS*, 2015.

[171] S. Soltesz, M.E. Fiuczynski, L. Peterson, M. McCabe, and J. Matthews. Virtual Doppelgänger: On the Performance, Isolation, and Scalability of Para-and Paene-Virtualized Systems, 2006.

[172] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable. In *Proc. of ACM EuroSys 2007*.

[173] Riley Spahn, Jonathan Bell, Michael Z Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-grained Data Management Abstractions for Modern Operating Systems. In *Proc. of USENIX OSDI*, 2014.

[174] P. St-Andre. Extensible Messaging and Presence Protocol (XMPP). *IETF Network Working Group, RFC3920*, 2004.

[175] S. Sukaridhoto, N. Funabiki, T. Nakanishi, and D. Pramadihanto. A Comparative Study of Open Source Softwares for Virtualization With Streaming Server Applications. In *Proc. of IEEE ISCE*, 2009.

[176] Kishor S Trivedi. Prepaging and Applications to Array Algorithms. *IEEE Transactions on Computers*, 1976.

[177] Jaideep Vaidya, Christopher W Clifton, and Yu Michael Zhu. *Privacy Preserving Data Mining*. Springer Science & Business Media, 2006.

[178] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proc. of SOSP, 2005*.

[179] Kun Wang, Jia Rao, and Cheng-Zhong Xu. Rethink the Virtual Machine Template. In *Proc. of ACM VEE 2011*.

[180] Markus Weiss, Adrian Helfenstein, Friedemann Mattern, and Thorsten Staake. Leveraging Smart Meter Data To Recognize Home Appliances. In *Proc. of PerCom*, 2012.

[181] A. Whitaker, M. Shaw, and S.D. Gribble. Scale and Performance in the Denali Isolation Kernel. *ACM SIGOPS Operating Systems Review*, 2002.

[182] A. Wolbach, J. Harkes, S. Chellappa, and M. Satyanarayan. Transient Customization of Mobile Computing Infrastructure. In *Proc. of MobiVirt, 2008*.

[183] Theodore M Wong and John Wilkes. My Cache Or Yours?: Making Storage More Exclusive. In *Proc. of USENIX ATC*, 2002.

[184] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proc. of NSDI 2007*.

[185] Wei Xu, Fangfang Zhang, and Sencun Zhu. Permlyzer: Analyzing Permission Usage in Android Applications. In *Proc. IEEE ISSRE*, 2013.

[186] M. Zeifman and K. Roth. Nonintrusive appliance load monitoring: Review and outlook. *Consumer Electronics, IEEE Transactions on*, 2011.

[187] Irene Zhang, Tyler Denniston, Yury Baskakov, and Alex Garthwaite. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *Proc. of USENIX ATC*, 2013.

[188] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. Fast Restore of Checkpointed Memory Using Working Set Estimation. In *Proc. of ACM VEE*, 2011.

[189] Pu Han Zhang, Jing Zhe Li, Shuai Shao, and Peng Wang. PDroid: Detecting Privacy Leakage on Android. In *Applied Mechanics and Materials*, 2014.

[190] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud Computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 2010.

[191] Jun Zhu, Zhefu Jiang, and Zhen Xiao. Twinkle: A Fast Resource Provisioning Mechanism for Internet Services. In *Proc. of IEEE INFOCOM*, 2011.

[192] Ahmed Zoha, Alexander Gluhak, Muhammad Ali Imran, and Sutharshan Rajasegarar. Non-Intrusive Load Monitoring Approaches for Disaggregated Energy Sensing: A Survey. *Multidisciplinary Digital Publishing Institute Journal of Sensors*, 2012.