IMPROVING THE SECURITY AND USABILITY OF CLOUD SERVICES WITH USER-CENTRIC SECURITY MODELS

BY SAMAN ZARANDIOON

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Danfeng Yao and Vinod Ganapathy

and approved by

New Brunswick, New Jersey

May, 2012

ABSTRACT OF THE DISSERTATION

Improving the security and usability of cloud services with user-centric security models

by Saman Zarandioon

Dissertation Director: Danfeng Yao and Vinod Ganapathy

Cloud computing is a paradigm shift in the way we define software and hardware, and architect our IT solutions. The emerging cloud technologies, due to their various unique and attractive properties, are evolving with tremendous momentum and rapidly being adopted throughout the IT industry. In this dissertation, we identify security challenges that arise in integration of cloud-based services, and present a set of novel solutions to address them. We analyze the security of our solutions, demonstrate their usage and effectiveness, and evaluate their performance by extensive experimentation. To address the problem of access control in untrusted cloud storage, we introduce *K2C* (Key To Cloud) protocol, which is a cryptographic access control protocol based on our new key-updating scheme referred to as *AB-HKU*. To improve the security and usability of integrated cloud services, we introduce a flexible client-side integration framework called *OMOS*. This framework enables secure and seamless client-side integration of cloud-based resources and services. Finally, to address the problem of identity management in an integrated cloud environment, we present a user-centric identity management solution called *Web2ID*. Our *Web2ID* protocol, by leveraging secure client-side cryptography and communication, introduces a privacy-preserving and secure mechanism for user authentication, fine-grained access delegation and identity attribute exchange.

Acknowledgements

First and foremost I would like to thank my co-advisors Danfeng Yao and Vinod Ganapathy for all their support, guidance and advice. They were always available and willing to patiently guide me throughout my PhD studies. During development of this work, I found their insight, suggestions, and encouragement precious.

I also would like to express my gratitude to the rest of my committee members Dr. Rebecca Wright, Dr. Ulrich Kremer, and Dr. William Horne for reviewing my work and their feedbacks.

I owe a great deal to all my teachers and peers in my undergraduate institute (BIHE). My most sincere thanks go to Mr. Kamran Mortezaei Farid, Mr. Mahmoud Badavam, Hootan Mahboubi, Rad Karimi, and Abbass Amirabadi for being a great source of inspiration for me.

Last but not least, I would like to express my eternal gratitude to my parents and sisters, Sanaz and Samira, for their everlasting love and support.

Dedication

I dedicate this work to Mr. Kamran Mortezaie Farid.

Table of Contents

Chapter 1

Introduction

In this chapter we define the scope, goal and thesis of this dissertation. We first define terms and concepts related to cloud computing and provide a taxonomy of cloud computing. Then we discuss new security and privacy challenges posed by cloud-based architecture, and summarize our contributions in addressing these challenges. Finally, we present the organization of the rest of this dissertation. We try to keep our definitions and terminology consistent with *The NIST Definition of Cloud Computing* [69].

1.1 Cloud Computing

Cloud computing is a computing model in which hardware, platform, infrastructure and software are defined and delivered as a service rather than a product. Cloud computing is emerging from recent advances in technologies such as hardware virtualization, Web services, distributed computing, utility computing and system automation.

Cloud computing takes advantage of hardware virtualization to securely and dynamically allocate physical resources such as computational power, storage, and networks to the users. Cloud resources are are delivered to the end-users through Web services. This simple model results in the following attractive features:

- *Elasticity*: Since physical resources are dynamically allocated to the consumers according to their needs, cloud services can scale on-demand.
- *Cost Effectiveness*: Resource sharing improves utilization of physical resources and thus reduces the associated cost.
- *Pay-as-you-go Pricing Model*: Cloud services have consumption-based metering and billing; this property makes them more affordable for small businesses and

startups.

- *Global-scale Accessibility and Usability*: Cloud consumers have access to a virtually unlimited physical resource pool through Web.
- *Easy Maintenance*: All non-functional requirements of IT, such as maintenance of hardware and software, are addressed by cloud providers, therefore consumers can concentrate on their functional business requirements.

1.2 Taxonomy of Clouds

To better understand the scope of cloud computing and related concepts and technologies, in this section we present a taxonomy of cloud.

1.2.1 Participants

In a cloud-model there are four main participants:

- *Cloud Provider*: A cloud provider (service provider) is an entity that is responsible for every thing required for making a cloud service available.
- *Cloud Consumer*: A cloud consumer is either a *cloud service owner* or a *cloud service consumer*. *Cloud service owner* is the individual or organization who subscribes for a cloud service. If there is any charge associated with the service, the cloud service owner will be responsible for the bills. *Cloud service consumer* is an individual or application who accesses a cloud service.
- *Cloud Broker*: A cloud broker is an entity that mediates between cloud providers and cloud consumers. The goal of a service broker is to provide the cloud consumer a service that is more suitable for its needs. This can be done by simplifying and improving the service and contract, aggregating multiple cloud services or providing value-added services. One can consider cloud brokers as a special cloud provider.

- *Cloud Auditor*: A cloud auditor is an independent party who examines a cloud service stack to provide an assessment on security, privacy and availability level of the corresponding cloud services and ensures that the corresponding SLAs (Service Level Agreement) are fulfilled. The details and scope of auditing process is normally specified in the service contract.

1.2.2 Cloud Services

The services provided by cloud providers can be divided into following three main layered categories. Each layer consumes services provided by the layer below it.

1. *Software as a Service (SaaS)*: All types of softwares including financial, CRM, HR, Sales, and office assistance can be delivered as a service. Salesforce.com, Google Docs, and Zoho Docs are some examples of SaaS services. Consumers of SaaS services, who are usually end users of the application or software administrators, access these types of softwares through web browsers or mobile apps.

2. *Platform as a Service (PaaS)*: Database, middleware, and integration bus are examples of platform resources that are provided by PaaS providers as a service. PaaS services are normally consumed by developers, testers, deployers, middleware/integration engineers and application administrators. Google App Engine is an example of a popular PaaS.

3. *Infrastructure as a Service (IaaS)*: IaaS clouds provide their consumers with low-level infrastructure resources, such as storage, Content Delivery Network (CDN), computational power, networks, backup and recovery, as a service. Typical IaaS consumers consist of system developers, network engineers, system administrators, monitoring engineers and IT managers.

1.2.3 Isolation Levels

With respect to deployment model and isolation levels, clouds can be categorized into the following five categories:

- *Public Cloud*: A public cloud is a cloud that its infrastructure is shared by many mutually untrusted cloud consumers.
- *Private Cloud*: If the infrastructure of a cloud is dedicated to a specific organization, we refer to that cloud as a private cloud. A private cloud can be on or off premise.
- *Virtual Private Clouds*: Off-premise clouds that are isolated from untrusted organization only through virtual network isolation (not physical network isolation) are called virtual private cloud.
- *Community Clouds*: Community clouds are clouds that their services are accessible to a particular set of organizations which form a community. Community clouds can all be on or off premises.
- *Hybrid Clouds*: A cloud that is a composition of two or more types of clouds is called hybrid cloud. These types of clouds are becoming increasingly more popular. Integration of these clouds poses some security challenges which we discuss in this chapter.

1.3 Cloud Integration

In traditional IT architecture enterprises use middleware technologies for internal or business to business (B2B) integration. In a cloud-based architecture, however, integration is more complex as it involves multiple heterogeneous environments and untrusted or semi-trusted entities. Therefore, cloud integration introduces new challenges and requirements that cannot be directly addressed by traditional middleware solutions. Below we list some common cloud integration scenarios:

- *Internal Cloud Integration*: Cloud has a layered architecture composed of three main layers: Application layer, Platform layer, and Infrastructure layer. Therefore, inside each cloud these layers should interact, integrate and collaborate with each other.

- *Cloud-to-Cloud (C2C) Integration*: In community clouds a set of heterogeneous or homogeneous clouds need to be integrated with each other.

- *Enterprise-to-Cloud (E2C) Integration*: For security and historical reasons, most of the confidential and corporate data needs to be maintained in enterprise data centers. This requires seamless integration of on-premise data and applications with public cloud services.

- *SaaS to SaaS/on-premise Integration*: Normally each SaaS provider services only a specific line of business. As a result, depending on its business requirements, an enterprise may need to use services from multiple SaaS providers. In this case, to fulfill a specif business process, these service providers may need to interact with each other. In these types of interactions real-time data synchronization and seamless connectivity is essential.

- *Interaction of Cloud Participants*: By definition, development, deployment, management and usage of cloud services requires interaction of different individuals, organizations and services. These entities need to be seamlessly integrated so that they can effectively and securely communicate.

Compared to traditional models, the diversity and distributed nature of cloud-based applications and infrastructure services makes the problem of cloud integration more complex. These challenges and complexities demand more versatile and adaptive integration solutions and technologies. In this dissertation, we introduce some protocols and frameworks to facilitate seamless communication between cloud entities by empowering them to securely interact with each other.

1.4 Security and Privacy Challenges in Clouds

In spite of its popularity, however, cloud computing has raised a range of significant security and privacy concerns which hinder its adoption in sensitive environments. The transition to cloud computing model exacerbate security and privacy challenges, mainly due to its dynamic nature and the fact that in this model hardware and software components of a single service span multiple trust domains. In the cloud, data and services are not restricted within a single organization's perimeter. This dynamism and fluidity of data introduces more risk and complicates the problem of access control [77]. Therefore, compared with the traditional models, in cloud computing model ensuring confidentiality and integrity of the end-users' data is far more challenging.

Moreover, cloud services are usually multi-tenancy services, meaning that a single infrastructure, platform, or software provides its services to multiple mutually untrusted parties simultaneously [31]. Therefore, confidentiality of these parties' data need to protected against each other. However, in some cases these parties may want to collaborate and share some data with each other in a controlled manner and thus there should be a mechanism that allows them to collaborate.

Layered architecture of cloud computing requires different levels of security considerations. In this work we are mainly concerned with the problem of identity management and access control in application and service level. We introduce a set of multi-party protocols specifically designed for cross-domain integrated cloud services. The main objective of these protocols is to provide more visibility and control to the end-users and close the gap between capabilities of existing solutions and new requirements of cloud-based systems.

1.5 Summary of Contributions

In this dissertation we introduce a set of protocols and frameworks that address some cloud security challenges introduced by the gap between the existing web security solutions and the security requirements of cloud consumers. Below we enumerate the main contributions of our work:

- We introduce a novel cryptographic access control protocol for untrusted cloud storage called *K2C*. Our protocol provides a flexible, privacy-preserving and secure way for cloud storage consumers to store and manage their data in the cloud without requiring them to fully trust the cloud provider.

- To enable seamless, dynamic and secure interaction of could services, we design and implement an integration framework called *OMOS*. Our framework facilitates secure client-side interaction of cloud services and gives the end-users more control over their protected resources.

- We present *Web2ID* protocol which is a user centric and privacy preserving identity and access management protocol. *Web2ID* is specifically tailored for integrated web and cloud environments.

1.6 Thesis

The flexibility, security and privacy of cloud services can be enhanced by adopting user centric access control and identity management solutions. Recent advances in cryptography and web technologies allow us to design security solutions that give the consumers more visibility and control over their cloud-based resources and thus alleviate some security and privacy concerns and fears associated with cloud paradigm. In other words, these solutions make it feasible for the enterprises to adopt cloud paradigm and outsource hardware, infrastructure and software while maintaining control over their identities and data.

1.7 Related Work

Cryptographic access control protocols are studied in the literature in the context of shared and untrusted file systems [60, 62]. However, these protocols suffer from a trade-off between granularity and scalability. Finding a cryptographic solution that is granular and scalable was an open problem until very recently that Shucheng Yu et al. in [88] introduced a novel approach that addresses this trade-off. However, in their protocol revocation requires re-encryption which is very expensive. In *K2C* protocol we address these restrictions by introducing a cryptographic access protocol that is granular enough to support hierarchies also scalable with respect to key management and revocation.

By providing cloud resources in the client-side, cloud providers can give their consumers more control and visibility over their cloud resources and identities. To facilitate integration with these types of cloud resources, recently different integration frameworks have been proposed [83, 58, 63]. However, these frameworks require change to browser, limit the functionality of service providers and consumers, or require a trusted component to ensure security and privacy of the framework. We introduce a client-side integration framework that without requiring any change to the browser or trusted component provide an infrastructure for secure interaction between client-side service providers, consumers and the end-user. On top of this framework we introduce an identity management solution that addresses some privacy and scalability concerns with existing protocols such as OpenID and OAuth. We will provide more details on specific relevant work in each chapter.

1.8 Organization of the Dissertation

This dissertation is structured as follows. Chapter 2 presents a solution for access control to untrusted cloud storage. Chapter 3 introduces an integration framework for secure and seamless interaction of cloud services. Chapter 4 presents a user-centric and privacy-preserving identity management and access control protocol for cloud mashups. Chapter 5 concludes with directions for future work.

Chapter 2

Client-side Access Control

In this chapter we present a user-centric privacy-preserving cryptographic access control protocol called *K2C* (Key To Cloud) that enables end-users to securely store, share, and manage their sensitive data in an untrusted cloud storage anonymously. *K2C* is scalable and supports lazy revocation. It can be easily implemented on top of existing cloud services and APIs – we demonstrate its prototype based on Amazon S3 API.

K2C is realized through our new cryptographic key-updating scheme, referred to as *AB-HKU*. The main advantage of the *AB-HKU* scheme its support for efficient delegation and revocation of privileges in hierarchies without requiring complex cryptographic data structures.

We analyze the security and performance of our access control protocol, and provide an open source implementation. Two cryptographic libraries, Hierarchical Identity-Based Encryption and Key-Policy Attribute-Based Encryption, developed in this project are useful beyond the specific cloud security problem studied.

2.1 Introduction

In industries such as health-care, insurance and financial organizations, which deal with sensitive data, the question of how to ensure data security and privacy in cloud environments is crucial [36, 82] and even of legal concerns. For example, in the health-care industry the privacy and security of protected health information (PHI) need to be guaranteed according to HIPAA (Health Insurance Portability and Accountability Act)[5] requirements.

To take advantage of public clouds, data owners must upload their data to commercial cloud providers which are usually outside of their trusted domain. Therefore, they need a way to protect the confidentiality of their sensitive data from cloud providers. Moreover, in many cases, data owners also play the role of content provider for other parties. Following the naming convention used in [84, 88], we refer to the parties that consume data owner's data as *data consumers* or *end-users*. For example, a healthcare provider (data owner) may need to let a medical doctor (data consumer) access medical record of his patient. In turn, a data consumer itself may recursively play the role of data owner. A medical doctor may want to share part of his patient's medical record with his secretary or nurse. Therefore, there is a need for a decentralized, scalable and flexible way to control access to cloud data without fully relying on the cloud providers.

In this chapter present design and implementation of a scalable, user-centric, and privacy-preserving access control framework for untrusted cloud storage. Our framework protects the confidentiality and integrity of stored data as well as the privacy of end-users. It is also implementable on top of existing cloud services and APIs. (Design goals in more details are presented in Section 2.3.1) .

Traditional access control techniques are based on the assumption that the server is in the trusted domain of the data owner and therefore an omniscient reference monitor can be used to enforce access policies against authenticated users. However, in cloud-based services this assumption usually does not hold and therefore these solutions are not applicable. Cryptographic access control techniques designed for shared/untrusted file systems are potential candidates for clouds. In these approaches, the data stored on untrusted storage is encrypted and the corresponding decryption keys are disclosed only to the authorized users. Therefore, the confidentiality of data is protected against untrusted storage as well as unauthorized users.

However, the existing solutions [53, 60, 62] have scalability limitations that hinder their adoption in the cloud-storage settings. For example, until recently finding a cryptographic approach that simultaneously supports fine-granularity, scalability, and data confidentiality was an open problem. In [88], Shucheng Yu et al. addressed this open problem by introducing a novel protocol which closes this gap.

Another scalability issue, which we address in this paper, is related to access revocation. To eliminate re-encryptions required as part of access revocation, a technique called *lazy revocation* is widely adopted by existing cryptographic filesystems [20, 78, 81]. Lazy re-encryption delays required re-encryptions until the next write access [1]. In practice, lazy revocation eliminates extra re-encryptions as write access requires the client to re-encrypt the data anyway. Therefore, lazy revocation significantly improves the performance at the cost of slightly lowered security. To support lazy revocation, cryptographic access control protocols need to use a *key-updating scheme* which provides *key regression*. Key regression enables a user holding a new key to derive an older key.

Despite the recent developments on untrusted cloud storage, current key-updating schemes are still inadequate in terms of usability and efficiency. Specifically, existing key-updating schemes [20], especially for access hierarchies, are not scalable as they require complex data structures such as cryptographic trees [53] or linked lists [62] (section 2.2). These cryptographic data structures need to be updated after each revocation. Since most of the existing cloud storage services have very simple APIs which allow only storing and updating key-value pairs, implementation of existing key-updating schemes on top of existing commercial clouds is inefficient and unscalable.

We introduce a new key-updating scheme called *AB-HKU* which is scalable and also supports access hierarchies without requiring complex data structures. Our *AB-HKU* scheme enables us to support lazy revocation without requiring any change to the existing cloud APIs. We also introduce a new signature scheme for Key-Policy Attribute-Based Encryption [51] called *AB-SIGN*. We then apply these new cryptographic schemes to achieve scalable and anonymous information sharing in existing commercial cloud storage services. We provide an implementation of the proposed protocols and perform extensive experimental evaluation on cloud storage environments. Our technical contributions are summarized as follows:

- We introduce a new scalable and secure key-updating scheme for access hierarchies.

[1]Lazy re-encryption, adopted by [88], delays re-encryptions till next (read or write) access. Since in regular workloads read accesses are significantly more than write accesses, the performance gain by lazy revocation is drastically more than that of lazy re-encryption.

- We design and implement a scalable and privacy-preserving access control framework for existing untrusted cloud services. Our framework supports lazy revocation and access hierarchies.

- We present a signature scheme for Key-Policy Attribute-Based Encryption [51]. Using our signature scheme, users can prove that they own a key that its policy satisfies with a set of attributes, without revealing their identity or credentials.

- We provide the first open source implementation of cryptographic libraries for Hierarchical Identity-Based Encryption [10] and Key-Policy Attribute-Based Encryption [12] schemes. They are useful beyond the specific cloud storage problem studied.

This chapter is organized as follows. In Section 2.2 we introduce our new key-updating scheme and prove its security. In Section 2.3 we explain our access control protocol and discuss its features and security guarantees. Then, in Section 2.4 and 2.5 we discuss the implementation details of our cryptographic libraries and access control framework and evaluate their performance. In Section 2.6 we discuss the related work. Conclusions and future work are given in Section 2.7.

2.2 Key Updating Schemes For Access Hierarchies

In this section we present an efficient secure key-updating scheme that supports hierarchies. First, we provide a formal definition for secure key-updating schemes for hierarchical access. Then, we give a concrete construction of a key-updating scheme based on the use of attribute-based encryption scheme. Our solution supports both key revocation and hierarchical delegation of secret access keys. Our secure cloud storage framework for easy sharing and revocation, described in Section 2.3, is built based on those two key properties.

2.2.1 Background

Lazy revocation, first introduced in Cephues [45], is a technique which reduces the overhead of revocation at the price of slightly lowered security [53]. When a user's read access right on a file is revoked, lazy revocation allows to postpone re-encryption of that file until the next change. Lazy revocation has been adopted by all majors cryptographic file systems [20, 78, 81]. However, it also causes fragmentation of encryption keys in access hierarchies. Therefore, a user receiving the most recent key of an access hierarchy should be able to compute the older keys in order to decrypt files that are not yet re-encrypted by the most recent key, a capability that is referred to as *key regression* [46]. *Key-updating schemes* [20] are cryptographic schemes which support *key regression*.

Another key management issue that we need to address is related to access hierarchies. A user owning access key of a specific hierarchy class should be able to decrypt all objects belonging to that hierarchy as well as all lower hierarchies. Key management schemes for hierarchies generate keys that satisfy this requirement. Key-updating schemes enable users to move backward in time dimension and decrypt data objects encrypted by older keys, whereas key management schemes for hierarchies let users traverse space forward and decrypt data objects encrypted by keys which correspond to lower hierarchies. Access control protocols that are coupled with folder structure of file system ([53]) and need support for lazy revocation, require schemes that let the users simultaneously traverse time backward and space forward. For example, a user holding the most recent version of an access key for folder `/a` should be able to decrypt a file located at `/a/b/c` which is encrypted by an older key.

In [20], Backes et al. formalize key-updating schemes. They also analyze and evaluate existing protocols that support key regression, but none of these protocols support hierarchies. In [26, 27], Blanton et al. formalize key management schemes for hierarchies, study existing protocols and introduce an efficient protocol for managing keys in hierarchies. But all of these schemes and protocols are static with respect to time, as they do not support key-updating/regression. *Therefore, none of these schemes are capable of handling key regression and hierarchies simultaneously.*

To our knowledge, the only work on key regression (lazy revocation) in hierarchies is [53], in which Grolimund et al. introduced the concept of Cryptree, a tree constructed by symmetric and asymmetric cryptographic links. In Cryptree, a user holding a clearance key pointing to a folder can traverse a sequence of cryptographic links to derive access keys to all of its sub-folders and files. Moreover, the structure of the Cryptree lets the protocol delay re-encryption of data until the next update; thus supports lazy revocation. However, for the reasons that we explain in Section 2.6, the complexity of required data structure and its high performance cost for large volume of data makes its implementation on top of existing cloud services unscalable and inefficient.

2.2.2 Model and Definitions in HKU Scheme

In this section we present a formal definition for Hierarchical Key Updating (HKU) Schemes and its security. Let $T = (V, E, O)$ be a tree that represent a hierarchical access structure. More general access class hierarchies in which partially ordered access classes are represented by a DAG are studied in [27]. In our work, we are only interested in a special case where DAG is a tree. Each vertex v_i in $V = \{v_0, v_1, ..., v_n\}$ corresponds to an access class. v_0 is the root and an edge $e = (v_i, v_j) \in E$ implies that v_i class is the parent of class v_j.

For example, *top secret*, *secret*, *confidential*, and *unclassified* form a hierarchy of access classes, where the root *top secret* access class is the parent of the *secret* access class. In a more complex access tree, a parent access class may have two or more child access classes. For example, a root *Enterprise* access class may have *Marketing*, *Manufacturing*, and *R&D* as its child access classes. We sometimes refer to access class as class, and use terms *node*, *vertex* and *access class* interchangeably.

O is a set of sensitive data objects, each object o is associated with exactly one access class $\mathcal{V}(o)$. In this model, any subject that can assume access rights at class v_i is also permitted to access any object assigned to a vertex that is a descendant of v_i.

The following definitions introduce the concept of time into our model.

Definition 1 *The local time at vertex v_i is an integer t_i that increases (elapses) every* *time access rights of a subject to that class is revoked.*

Definition 2 *The global time associated with node v_i is a vector $\tau_i = (t_0, ..., t_j, .., t_i)$ where t_j is the local time of j^{th} vertex on the path from root to vertex v_i on the access tree T.*

Two instances of global time are comparable only if the vertices that they belong to are identical or one of them is the ancestor of the other one; We say $\tau_i < \tau_j$ iff τ_i and τ_j are comparable and all common components of τ_i are less then the corresponding components in τ_j . Similarly, we define comparative operators $=$, $>$, $\leq$, and $\geq$.

Definition 3 *A Hierarchical Key-Updating (HKU) Scheme consists of a root user and end users. An end user may be a reader, a writer, or both. There are five polynomial time algorithms HKU = (Init, Derive, Encrypt, Decrypt, Update) defined as follows.*

- *Init*$(1^k, T)$ *is a randomized process run by the root user which takes as input a security parameter k and an access hierarchy tree T and then generates and publishes a set of public parameters Pub and outputs the root key $K_{v_0,\perp}$. It also initializes the state parameters including the value of local time at each vertex.*

- *Derive*$(T, K_{(v_i,\tau_i)}, v_j)$ *is a randomized process run by the root user, reader or writer which using the private key $K_{(v_i,\tau_i)}$ of v_i at time τ_i derives a private key of target class v_j at its current global time τ_j according to T. Derive computes the requested key only if v_i is an ancestor of v_j and $\tau_j = \tau_i$; otherwise, it outputs null ($\perp$).*

- *Revoke*(T, v_i) *run by the root user, reader or writer, increments the local time t_i of v_i by one, updates other state variables, and returns the updated tree T'.*

- *Encrypt*(T, o_k) *is a randomized algorithm called by writer that encrypts the data object o_k and returns the encrypted object C.*

- *Decrypt*$(K_{(v_i;\tau_i)}, C)$ *is a deterministic process run by reader which takes a key and an encrypted object as inputs and returns the corresponding object in plaintext. This function can decrypt C only if it belongs to the same or a descendant of the* *access class that the key belongs to and the time that o_k is encrypted at is less than or equal to τ_i; otherwise, it outputs null ($\perp$).*

Definition 3 is a generalization of the definition of key-updating schemes in [20] and the definition of key allocation schemes for hierarchies in [27]. If we assign to T a tree of depth 1 where its leaves are a set of groups (i.e, remove hierarchies), our definition reduces to a key-updating scheme defined in [20] and if we remove the update process and the time dimension, our scheme reduces to key allocation scheme for hierarchies defined in [27]. Intuitively, a hierarchical key-updating scheme is secure if all polynomial time adversaries have at most a negligible advantage to break the ciphertext encrypted with the current-time key of a target class, assuming that the adversaries do not belong to higher (ancestor) target classes in the hierarchy, or possess keys for earlier time periods. In this model the adversary chooses her target at the beginning of the game and then adaptively queries the scheme.

We define the security model of hierarchical key-updating schemes as follows:

Definition 4 *A hierarchical key-updating scheme is secure if no polynomial time adversary $\mathcal{A}$ has a non-negligible advantage (in the security parameter k) against the challenger in the following game (HKU game) :*

Choosing target: *The adversary declares an access object $\hat{v}$ and a time instance $\tau_{\hat{v}}$ that she wishes to guess its corresponding private key (i.e. $K'(v', \tau')$).*

Setup: *The challenger runs Init$(1^k, T)$, and gives the resulting public parameters Pub and T to the adversary.*

Key-Extraction Query: *The adversary adaptively queries the private keys of polynomial number of vertices at any time that she wishes subject to the restriction that either the queried vertices are not an ancestor of (or equal to) $\hat{v}$ or the time instance that they are being queried at is earlier than or equal to $\tau_{\hat{v}}$.*

Challenge: *The adversary submits two equal length objects o_0 and o_1 belonging to the access class $\hat{v}$. The challenger flips a random coin b, and encrypts o_b for time $\tau_{\hat{v}}$ and submits the result to the adversary.*

The adversary issues more Key-Extraction queries.

Guess: *The adversary outputs a guess b' of b.*

Adversary's advantage is the probability that her guess is correct: $Adv_{\mathcal{A}} = Pr[b' = b]$. The HKU scheme is secure if the adversary's probability compared to random guessing ($\frac{1}{2}$) is negligible.

2.2.3 AB-HKU Scheme

In this section, we present a concrete construction for *HKU* scheme called *AB-HKU*. This scheme is based on the use of bilinear map and the difficulty of the Decisional Bilinear Diffie-Hellman problem. Our solution is realized on top of the Key-Policy Attribute-Based Encryption scheme (KP-ABE) [51] and invokes KP-ABE operations including SETUP_ABE, KEYGEN_ABE for private key generation, ENCRYPT_ABE for data encryption, and DECRYPT_ABE for decryption.

- *Init*$(1^k, T)$

  1. The root user runs the SETUP_ABE process with 1^k as security parameter to generate ABE public parameters and the master key MK. Publishes the ABE public parameters as $\mathrm{Pub}_{\mathrm{abe}}$.
  2. Calls KEYGEN_ABE procedure using MK as the secret key and "$L_0 = v_0$" as its policy. Outputs the result as the root key ($K(v_0, \perp)$= KEYGEN_ABE(MK, $L_0 = v_0$)).
  3. To each vertex in T adds a local time variable t_i initialized to zero.

- *Derive*$(T, K(v_i; \tau_i), v_j)$ is run by a user (root user, reader, or writer) with secret key $K(v_i; \tau_i)$ at time τ_i to obtain the private key for node v_j.

  If class v_j is not a descendant of class v_i, or the time τ_i is not equal to current time τ_j associated with v_j, then return null. Otherwise, denote $(u_1, u_2, ..., u_n)$ as the list of vertices in the path from v_i to v_j; denote $(t_{u_1}, t_{u_2}, ..., t_{u_n}, t_{v_j})$ on T as the list of current local time values of intermediate vertices (including v_j); and let d represent the depth of v_i.

  The user performs the following operations.

1. Construct the access tree $\mathcal{T}'$ which corresponds to the following Boolean expression: ($L_d.v$ attribute represents vertex in d-th level, $L_d.t$ represents its current local time and $\wedge$ is conjunction operator.)

$$\begin{aligned}(L_{(d+1)}.v = u_1 \wedge ... \wedge L_{(d+n)}.v = u_n \\ \wedge L_(d+n+1).v = v_j) \wedge \\ (L_{(d+1)}.t \leq \tau_{u_1} \wedge ... \wedge L_{(d+n)}.t \leq \tau_{u_n} \\ \wedge L_{(v_j)}.t \leq \tau_{v_j}) \qquad (2.1)\end{aligned}$$

This Boolean expression restricts access to objects that belong to node v_j or its descendants and are created at current time or before.

2. Denote the access tree of $K_{(v_i,\tau_i)}$ by $\mathcal{T}$. Using the procedure for delegation of private key in [51], add the access tree $\mathcal{T}'$ to the root of $K(v_i, \tau_i)$, increase its threshold by one, update and calculate the private parameters associated to the root according the protocol. In implementation section we provide more details on this procedure.

3. Output the resulting access tree and parameters as a private key $K(v_j, \tau_j)$ for v_j.

- *Encrypt*(T, o_k): Denote v_i as the access class that object o_k belongs to. ($v_i = \mathcal{V}(o_k)$). Denote $(v_0, u_1, u_2, ..., u_n, v_i)$ as v_i's path and $\tau_i = (t_{v_0}, t_{u_1}, t_{u_2}, ..., t_{u_n}, t_{v_i})$ as its current time according to T. A writer encrypts o_k as follows.

   1. Set the attribute set γ as follows. The attribute set is used as the public key for encryption.

$$\begin{aligned}\gamma = \{L_0.v = v_0, ..., L_n.v = v_n, L_{n+1}.v = v_i, \\ L_0.t = t_{v_0}, ..., L_n.t = t_{u_n}, L_{n+1}.t = t_{v_i}\} \qquad (2.2)\end{aligned}$$

   2. Use ABE encryption procedure to encrypt o_k with attribute set γ and return the resulting encrypted object. ($C = \text{ENCRYPT\_ABE}(\text{Pub}_{\text{abe}}, \gamma, o_k)$).

- *Decrypt*$(K_{(v_i,\tau_i)}, C)$. The reader decrypts as follows.

1. If the encrypted object C does not belong to the same access class v_i as the key $K_{(v_i,\tau_i)}$ or one of its descendants, or the time when C is encrypted is later than the time τ_i when the key is generated, then return null ($\perp$).
2. Otherwise, run ABE decryption procedure and return its result as output (o_k= DECRYPT_ABE($K_{(v_i,\tau_i)}, C$)).

- *Revoke* (T, v_i) is run by a user to increment the local time of v_i by one and then returns the updated tree T'.)

The correctness of our HKU scheme follows the correctness of the key policy ABE scheme [51].

Theorem 1 *Assuming the hardness of the Decisional BDH, AB-HKU is a secure hierarchical key-updating scheme.*

Proof 1 *Sketch. It suffices to show that an adversary who can play HKU game for AB-HKUwith non-negligible advantage, can also win KP-ABE game with a non-negligible probability, and thus break the security of KP-ABE and subsequently the Decisional BDH. Let $\mathcal{A}$ be an adversary who can win HKU game with non-negligible advantage $\frac{1}{2}+\epsilon$. She can play KP-ABE Selective-Set model game as follows.*

Init: *$\mathcal{A}$ declares the set of attributes that corresponds to vertex $\hat{v}$ and time $\tau_{\hat{v}}$ as γ, the set of attributes that she wishes to be challenged uppon.*

Setup: *This step is identical to Setup step in HKU game.*

Phase 1: *In this phase the adversary queries for the private keys for access structures (trees) $\mathcal{T}_j$ which correspond to that of keys that she would query in HKU game. Since, according to the protocol of HKU game, these keys belong to vertices that are not an ancestor of $\hat{v}$ or their time is less than $\tau_{\hat{v}}$, their access trees will not satisfy with attributes in γ ($\gamma \notin \mathcal{T}_j$) and therefore they are legitimate queries.*

Challenge: *Identical to the Challenge step in HKU game.*

Phase 2: *Repeat Phase 1.*

Guess: *The adversary guesses b using the same strategy that she uses in HKU game.*

Since the data is encrypted under the same set of attributes and using the same procedure, she has the same non-negligible advantage to make the correct guess. This concludes our proof. □

2.3 K2C Protocol

We describe the application of our hierarchical key-updating scheme in realizing a secure and scalable cloud access control protocol that supports easy sharing and revocation on hierarchically organized resources. We also analyze the security of our protocol.

2.3.1 Design Goals

Below we list the design goals of our *K2C* access control protocol:

- **Security.** Our protocol must protect the confidentiality and integrity of stored data against cloud providers and unauthorized end-users. Meaning that the stored data should be readable for authorized users only and any unauthorized change to the data should be prevented or detectable.
- **Privacy-preserving.** Access rights of a specific end-user as well as his usage trends should not be visible to other users or cloud service providers.
- **Efficiency and Scalability.** To avoid unjustified cost of re-encryption, the protocol should support lazy revocation. Also, the complexity of operations should be independent of number of data objects and users in the system. This ensures that the protocol will not affect the scalability of existing cloud services.
- **Flexibility.** The protocol should allow data owners and end-users to organize and manage their data in hierarchies similar to conventional file systems. Directories also represent access class hierarchies, users who have access to a directory/folder also assume the same access to all files and directories below that directory. Also, they should be able to grant/revoke part of their access rights to/from other users in a decentralized and scalable manner.

- **Simplicity and Extensibility.** Last but not the least, the protocol should be simple enough to be efficiently implementable on top of existing commercial cloud APIs.

We assume end-users have secure communication channels, limited computation power and storage required for authenticating each other and performing client-side key distribution in a synchronous or asynchronous manner.

2.3.2 Security Model

In this section we present the security model for *K2C*. We assume that the root user, representing the data owner, is trusted. The cloud providers are honest-but-curious (aka semi-honest), who follow the protocol and faithfully execute the operations, but may actively attempt to gain additional knowledge, such as the sensitive data stored in the cloud. An adversary may attempt to perform unauthorized read or write access against the stored data, or attempt to learn the identities of readers or writers. For example, end-users may try to perform unauthorized read or write operations on stored data objects. To perform their attacks, unauthorized users may use their existing access keys for other objects and categories or cooperate with other unauthorized users and cloud providers to derive/guess credentials required to perform unauthorized access. Similarly, cloud providers may try to read or modify stored data or learn about the identities of the end uers. Cloud providers may collude with each other or some unauthorized end-users to break the security of *K2C*. We assume communication channels between participants are secure (e.g., SSL).

2.3.3 A signature scheme for KP-ABE

K2C requires a signature scheme to 1) enable the readers to verify the integrity of data and ensure that it is produced by an authorized writer, 2) enable the cloud service providers to validate incoming requests and block unauthorized accesses. However, the original paper which introduces KP-ABE [51] does not present any signature scheme. In this section we introduce an attribute-based signature scheme called *AB-SIGN* which enables the verifier to ensure that a signature is produced by a user whose access policy is satisfiable by a set of attributes without learning the signer's identity.

Our design for *AB-SIGN* is based on the same technique introduced by by Moni Naor (Section 6 of [30]) for Identity Based Encryption and then extended in [47] for HIDS signature scheme.

Definition 5 *AB-SIGN is a signature scheme for key-policy attribute-based encryption that its signing and verification methods are defined as follows. Let's say that the signer has a key K for policy P, and wants to sign message $\mathcal{M}$. The verifier needs to verify that the signature is generated by a signer whose key policy satisfies attribute set A:*

Signing

From K derive a key (K') which corresponds to a policy which is the concatenation of P and ($@S = \mathcal{M}$) ($@S$ is a reserved attribute for signatures). Send the derived key to the verifier as the signature.

Verification

Generate a random token and encrypt it using the attribute set $A \cup \{@S = \mathcal{M}\}$ and then decrypt the result using a key which is equal to the signature. If the result is equal to the original token the signature is valid (i.e. the attribute set A satisfies the signer's key policy.)

To prevent an attacker from using the signature method to derive a valid access key, we need to reserve the attribute '@S' for signature.

Theorem 2 *Assuming the hardness of the Decisional BDH, AB-SIGN is a secure signature scheme.*

Proof 2 *Sketch. Unforgeability of AB-SIGN scheme follows immediately from the security of KP-ABE scheme. In AB-SIGN, a signature is a derived key from the actual write access key. Therefore, based on the security of KP-ABE derive operation, the only entity who can generate the signature is the owner of the write access key. Moreover,* *security of derive operation guarantees that the verifier cannot guess the original access key from the derived key.* □

2.3.4 Protocol Description

In this section we provide the details of *K2C* protocol. The protocol runs between the root user, end-user (reader or writer), and the cloud providers. The root user may be a system administrator in the data owner's organization, who can specify the access privileges of end-users. The end-users may further delegate their access privileges to other individuals for easy sharing. We achieve the revocation of privilege by encoding the validity period in the private keys of users and advancing time with respect to the target hierarchy or data object. Another advantage of our K2C framework for use in cloud storage is the support of anonymous access.

As illustrated in Figure 2.1, *K2C* requires three repositories: *Meta-data Directory*, *Data Store* and *Key-store*.

- **Meta-data Directory**: All meta-data associated with hierarchies and data objects are maintained in this repository. *K2C* requires two properties for each object: *Read Access Revision* (RAR) and *Write Access Revision* (WAR). These two properties play the role of local time in *AB-HKU* for read and write access, respectively. In order to compute Read/Write Access Revision Vectors (which correspond to global time instances in *K2C*), the cloud provider that hosts Meta-data Directory needs to provide an API for querying RAR and WAR values of multiple directories in a single request. All existing cloud-based databases (also known as 'NoSQL systems', or 'schema-free database') such as Amazon SimpleDB [7], Microsoft Azure SQL [16], and Google's AppEngine [8] database (Bigtable [35])) satisfy this requirement and therefore qualify to host a *K2C* Meta-data Directory. For our experiments we use Amazon SimpleDB [7].

- **Data Store**: This repository contains the actual content of each data object. Any cloud key-value based storage system such as Amazon S3 [6] can be used as *K2C* Data Store. In *K2C*, *keys* are hierarchical path name of data objects

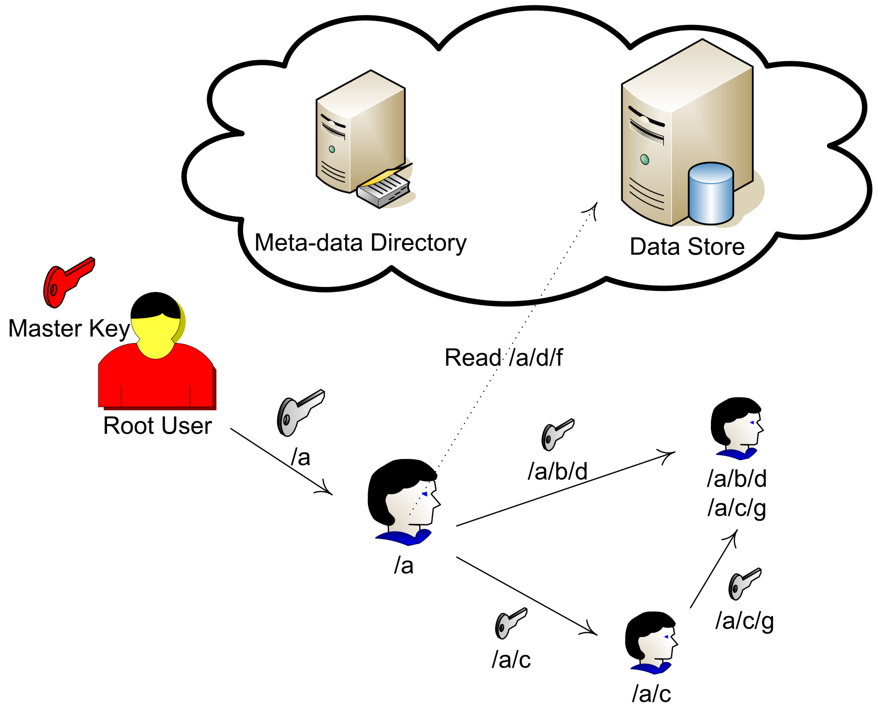

Figure 2.1: Illustration of all major participants of *K2C*. Following *K2C* protocol, end-users can enforce access control on their own data without fully trusting or relying on the cloud providers. In *K2C*, keys are distributed and managed in a distributed fashion. Solid arrows represent access delegation.

and *values* are the actual content of corresponding data objects. Cloud key-value storage providers are tuned for high throughput and low storage cost; these features make them a good candidate for *K2C* Data Store [2].

- **Key-store**: All read/write access keys of end-users are kept in their secure local repository called Key-store. Each key-store contains all public parameters as well as read/write access key entries of all data-objects and categories that the end user has access to. Each access key entry contains the following fields: object path, access type (read/write), granter's identity, and secret key. The Key-store provides an API that given user's credential and a path, returns the first key entry that its path is a prefix of the input path.

[2]Note that using key-value storage for Meta-data Directory is not efficient as computing WAR/RAR vector leads to multiple calls to the cloud storage system.

Initial Setup and Basic Operations

To setup *K2C*, the root user needs to follow the steps listed below:

1. Sign up for cloud services required for hosting Meta-data Directory and Data Store.
2. Run *Init* procedure according to *AB-HKU* scheme to generate public parameters and the master key.
3. Save the master key and public parameters in the root's Key-store.
4. Share the public parameters with the cloud service providers that support *K2C* request authorization.
5. Create an entry in Meta-date Directory that corresponds to the root directory. The WAR and RAR numbers of the root directory entry are initialized to zero.

There are four basic operations in our protocol: write, read, delegate, and revoke. Each basic operation leads to calls to Meta-data directory and/or Data Store. We present the high-level steps involved in these operations below. Other operations such as create/remove/rename for directories and data objects can be defined similarly. K2C requires that each request be signed by user's access key for the target object using our AB-SIGN operation. This requirement enables cloud providers which support K2C to block unauthorized request. We refer to this property as K2C request authorization.

- **Write**: To write into a specific data object, the end-user needs to perform the following steps (See also Figure 2.2).
  1. Retrieve the required write access key from the local Key-store.
  2. Query Meta-data directory to get read access revision (RAR) vector of the target object.
  3. Using *AB-HKU* scheme, encrypt the data by the retrieved RAR vector and its path.
  4. Using *AB-SIGN* scheme, sign the data by his write access key.

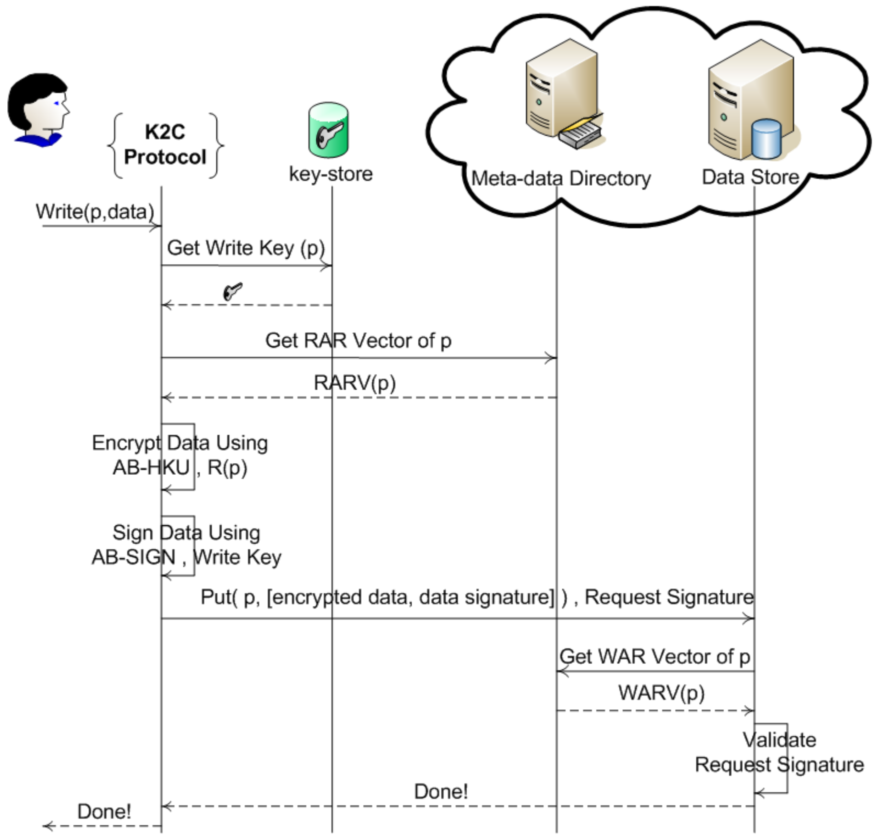
K2C Protocol
key-store
Meta-data Directory
Data Store
Write(p,data)
Get Write Key (p)
Get RAR Vector of p
RARV(p)
Encrypt Data Using AB-HKU , R(p)
Sign Data Using AB-SIGN , Write Key
Put(p, [encrypted data, data signature]) , Request Signature
Get WAR Vector of p
WARV(p)
Validate Request Signature
Done!
Done!

Figure 2.2: Write Operation.

5. Construct a key-value pair where the key is equal to the path of data object and the value is the encrypted data and corresponding signature. Store the pair in Data Store.

6. To prevent destructive writes by unauthorized users, the Data Store can query write access revision (WAR) vector of that object from the Meta-data Directory to validate the signature of request [3].

- **Read**: To read a specific data object stored using *K2C* protocol, the end-user needs to do the following (See also Figure 2.3). To ensure the data is produced by an authorized writer, the reader needs to validate the corresponding signature using *AB-SIGN* signature scheme. Then the reader can decrypt the data using its read access key and *AB-HKU* scheme.

[3]Note that unauthorized writes are also detectable by the reader.

1. Retrieve the required read access key from the local Key-store.
2. Using *AB-HKU* scheme and the read access key, decrypt the encrypted data.
3. Using *AB-SIGN* signature scheme, validate the signature to ensure that data is produced by a user who has the proper write access.
4. Return the decrypted data.

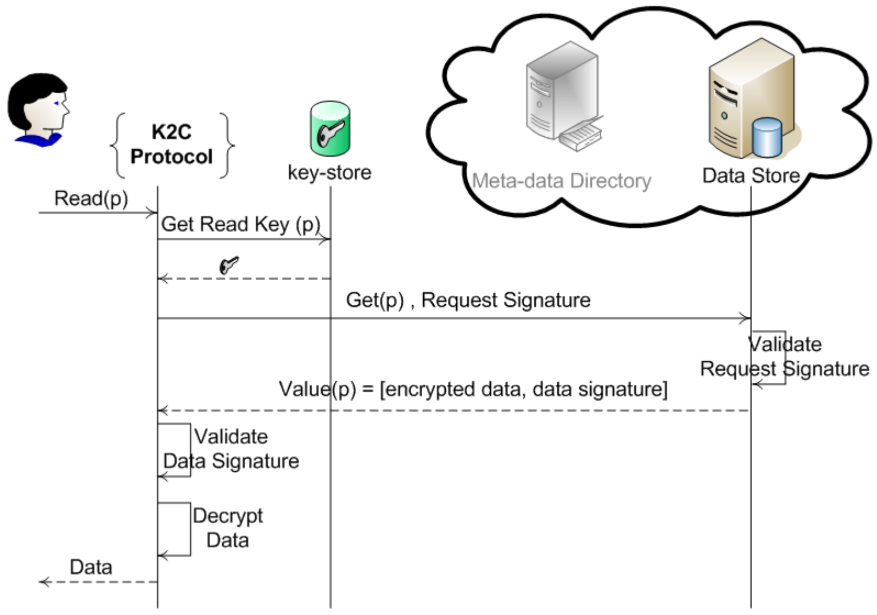

Figure 2.3: Read Operation.

- **Delegation**: Delegation operation can be run by a user to authorize another user a subset of his access privileges. It requires three input parameters: the identity of the delegatee, the resource path, and access type (read/write). The steps required for this operation are listed below:

  1. From the local Key-store, get the access key that matches the target resource path and access type.
  2. Query Meta-date Directory to get the read/write access revision (RAR/WAR) vector of target resource.
  3. Run *Derive* operation, as defined in *AB-HKU* scheme, to generate the required access key.

4. Send the generated access key to the delegatee through a secure communication channel.

- **Revocation**: To revoke a user's access on a specific directory or data object, the authorized user needs to make a signed request to the Meta-data Directory to increase the corresponding access revision number. To ensure the integrity of access revision numbers, these entries should be signed by the requester.

Key Distribution and Management

In *K2C*, there is no centralized entity responsible for key distribution and management. Each user is responsible for maintaining and managing its own access keys. Key owners can derive access key to all sub categories according to *AB-HKU*scheme and distribute them to other users thorough secure communication channels. *K2C*assumes there is already some secure communication channels between all entities so that they can identify each other and securely communicate in synchronous or asynchronous manner.

2.3.5 Security Analysis

In this section we state the security guarantees provided by *K2C* protocol. More detailed proofs and analysis can be found in our full version [89].

Confidentiality. Our solution ensures that only the users who have the most recent version of the access key of the data object or one of its ancestor directories can decrypt it. The confidentiality of stored data is protected under our protocol because writers always encrypt the data objects by their path and most recent read access revision (RAR) vector according to *AB-HKU* scheme. The cloud provider or other unauthorized users cannot gain any information that helps them to guess the access key of unauthorized data objects.

Collusion-resistance. Security of KP-ABE guarantees that unauthorized users and malicious cloud service providers cannot collude to guess access key to an unauthorized data object.

Integrity. The integrity of stored data is preserved. This guarantee is realized by requiring writers to sign the data by their write access key using *AB-SIGN* scheme. We require readers to validate writer's signature to ensure that it is produced by an authorized writer (i.e. a user with write access to that data object or on of its parent directories). Because meta-data entries stored in the Meta-data Directory are also required to be signed by the end-users, any unauthorized change in Meta-data Directory is detectable by the reader.

Anonymity. The end users are anonymous to each other and to the cloud providers. The signatures used in the our authorization do not contain any identify information. During the course of protocol, the end-users do not reveal any information about their credentials. *AB-SIGN* signatures bound to the data objects and requests, include only attributes related to the location and global time of those objects.

2.4 Our KP-ABE Crypto Library

To support lazy-revocation and hierarchies, *K2C* uses our *AB-HKU* scheme that is based on Key-Policy Attribute-Based encryption scheme [51]. But, we were not able to find any implementation of KP-ABE [4]. Therefore, we develop a general KP-ABE cryptographic library and release it as an independent open source project [12]. In this section, we provide a short overview of this library.

2.4.1 High-level Design

Our library implements the KP-ABE scheme. KP-ABE is a *large universe construction*, meaning that it does not require the attributes to be fixed during the initialization process. However, the maximum number of attributes should be known in advance – a limitation which is not desirable in many practical cases. To overcome this limitation, we adopt the random oracle model [23] and replace function $T(X)$ (used in the *Setup* phase) by a secure hash function. This modification also improves the efficiency of the

[4]The open source implementation of Ciphertext-Policy Attribute-Based Encryption (CP-ABE) [24] which was presented in the original paper [24] is available at [15]. However, CP-ABE is not applicable in our protocol.

library. Therefore, our library does not put any limitation on the number of attributes that can be used in the system. We support numerical attributes and comparisons [24].

In this approach, numerical attributes are converted into multiple symbolic attributes each representing a specific bit in its binary representation (a 'bag of bits' representation). Similarly, numerical comparisons are translated into an access tree which its leaves are the corresponding bit-wise attributes and evaluates to the same results as the numerical comparison. In other words, this approach converts numerical comparisons into a symbolic comparison.

In our library, policies are defined recursively and represented using an S-Expression (LISP-like expression) as follows:

$$\begin{cases} a = v & a \text{ is a symbolic attribute} \\ (c\ a\ v) & a \text{ is a numerical attribute \&} \\ & c \text{ is a comparative operator} \\ ([\ t \mid \texttt{and} \mid \texttt{or}\ ]\ p_1\ p_2\ \ldots\ p_n) & \text{Composite policy} \end{cases}$$

The first type corresponds to a simple policy for symbolic attributes. For example, policy `role=manager` gets satisfied only if the attribute `role` assumes a value equal to the literal `manager`. The second type represents policies for numerical attributes. For example, policy `(> age 18)` gets satisfied only if the value of the attribute `age` is greater than `18`.

The third type represents policies which are composed of a set of policies (i. e. $p_1, p_2, \ldots, p_n$) proceeded by a threshold. Composed polices get satisfied only if the number of satisfied polices in that list is more than or equal to the specified threshold. The threshold can be one of the following three items: an integer threshold $t \in [1, n]$, `and`, `or`. For example, `(and role=manager (> age 18))` is a composite policy which gets satisfied only when the value of attribute `role` is equal to `manager` and the value of the numerical attribute `age` is greater than `18`.

2.4.2 Implementation Details

Our library consists of two main public classes: `ABE` and `Entity`. `ABE` class can be used by the root user to initialize the system. It also represents the public parameters of the system. `Entity` class represents privileges of users within the corresponding system. `ABE` has a constructor that receives two security parameters (`rBits` and `qBits` with default value of 160 and 512, respectively) and generates all public parameters as well as the master key. The root user constructs an object of this class during the setup process and then calls a function called `ABE.getRootEntity()` to generate an instance of the `Entity` class which represents the root privilege. This class also provides a the methods for encrypting data using a set of attribute-values based on the public parameters. Moreover, it has a method for validating signatures using *AB-SIGN* signature scheme, which we introduced in this paper. This method validates that data is signed by an entity which its access policy satisfies with input parameters.

`Entity` class does not have any public constructor. The first instance of `Entity` is generated by calling `ABE.getRootEntity()` during the setup time (as explained in the previous paragraph). Then other instances will be derived from root and other instances. The `Entity.derive` method accepts a policy as input and generates an instance of `Entity` which its access tree corresponds to the conjunction of the input policy and the original policy. This means that the derived entities always have more restrictive access. The `derive` method is implemented using 'Re-randomization' and 'Converting a (t, n)-gate to a $(t+1, n+1)$-gate' operations as they are defined in [51]. During the derive operation we perform some optimization to keep size of the resulting access tree minimal.This class also provides methods for decrypting and signing data.

`ABE` and `Entity` classes are serializable. This feature enables us to store, retrieve, and transfer the public and private parameters. Moreover, encryption, decryption, and signature methods are steam-based meaning that their methods can process data in an `InputStream` and write the result in an `OutputStream`.

To improve the performance, the actual data is encrypted using a symmetric-key encryption scheme [5] and only the key is encrypted using KP-ABE.

KP-ABE is a pairing-based crypto-system which requires pairing-based operations such as elliptic curve generation, elliptic curve arithmetic and pairing computation. For pairing-based operations, we used a Java-based library called jPBC [11]. This library comes in two flavors: Java Porting, Java Wrapper. In our implementation and experiments we used the Java Wrapper version. Since in Java Wrapper version, the computation is done in C, it is faster but platform dependent. To parse a policy and construct an access tree ($\mathcal{T}$), we used jSExp [13], an open source Java library for parsing Lisp S-expressions.

We also implemented a similar crypto library for Hierarchical Identity-Based Encryption scheme [47]. The source code is accessible at [10].

2.5 K2C Framework

In this section we present the high-level architecture of *K2C* framework and explain the implementation details of its major components. We also provide some experimental results that show performance overhead of our solution.

2.5.1 Design and Implementation

Simplicity and extendability are two major design goals of our framework. To make *K2C* framework independent of any specific cloud provider, we abstract away the details of the cloud providers through two simple interfaces: `IDataStore` and `IMetadataDirectory`. A new cloud service provider can be easily supported by implementing these interfaces. To support a new cloud service provider, we need to do the following:

1. Implement these interfaces using the API of the new cloud service provider. We refer to their implementation as the *K2C*driver of that service provider.

2. Inject the new driver into *K2C*framework by binding it to the corresponding interface. This can be done easily through the framework's dependency provider

[5]The default is Advanced Encryption Standard (AES) [39] with the key size of 128 bits; however, it can be replaced by other symmetric-key encryption scheme.

module (A Google Juice [9] module).

Out of the box, *K2C* framework comes with a data store driver for Amazon S3 and a meta-data directory driver which uses Amazon SimpleDB. To make it easier for the developers to learn and use our framework, we expose its services through a set of APIs which are very similar to the Java APIs for accessing the file system. In fact, we first extracted the Java source code of `java.io.*` classes from the OpenJDK [70] project and then, by re-implementing the abstract class `java.io.FileSystem`, replaced all the calls to the filesystem with a call to the corresponding operations in the *K2C*protocol. Our framework uses a password-based encrypted [6] keystore similar to JKS (Java KeyStore). At runtime, the required keys are retrieved from the keystore and cashed into the memory. The users need to perform their key management operations manually using a command-line utility called `k2ctool`. For example, to delegate another user access to a specific folder (e.g. /Enterprise/IT/Middleware/docs/), the user needs to run the following command:

Listing 2.1: Delegation

```
k2ctool -delegate -object /Enterprise/IT/Middleware/docs/ \
                  -keystore /home/k2c/.keystore \
                  -storepass **** \
                  -output /home/${USER}/.key
```

This command generates the required access key and saves it into `/home/${USER}/.key`. Then the user needs to send the generated file to the delegatee. Later, using the `import` option of `k2ctool`, the delegatee can add this key into his own keystore . Similarly, during the initial setup, the root user can use the option `genkey` to generate the public parameters and the master key.

2.5.2 Performance Evaluation

In this section we provide some experimental results which show the performance overhead of our *K2C* protocol. We ran our experiments on a machine with the following

[6] PBE with Triple DES

configuration: Intel Pentium(R) Dual-Core CPU, 3.20GHz, Cache 2048 KB, 3.3 GB RAM, i386 GNU/Linux 2.6.

Pre-computation and caching. As discussed in the previous sections, to overcome the limitation of fixed attributes, we adopted *large universe construction* of KP-ABE. However, in this construction the process of mapping an attribute to the bilinear group $\mathbb{G}1$ (i.e. $\{1, 0\}^* \rightarrow \mathbb{G}1$) is very expensive (on average 23 ms per attribute). In our KP-ABE library every bit of a numerical attribute gets translated into a symbolic attribute. For example, a 10-bit representation of the numerical attribute $li = 355$ gets translated into a list of symbolic attributes shown in 2.2.

Listing 2.2: Symbolic representation of attribute $li = 355$

```
[li@0=1, li@1=1, li@2=0, li@3=0, li@4=0,
                        li@5=1, li@6=1, li@7=0, li@8=1, li@9=0]
```

Also, all numerical comparisons get translated into symbolic matching policies. For example, policy 2.3 corresponds to the numerical comparison $li < 358$.

Listing 2.3: KP-ABE policy for $li < 358$

```
(2 li@9=0 (1 (2 li@7=0 (1 (1 (2 li@4=0
                               (2 li@3=0 (1 li@1=0 li@2=0))
                          ) li@5=0) li@6=0)) li@8=0))
```

In *K2C*, every level of an object's path has two numbers associated with it - read access revision number and write access revision number. Therefore, these numerical attributes lead to many symbolic attributes which their mapping cost create a significant over-head. Since the value of each bit is either zero or one, we pre-compute the mapped values of these symbols and during the startup process load them into the framework. Moreover, at runtime, we cache the mapped value of each path segment in a hash table so that it can be reused. Using the described pre-computation and caching techniques, we were able to significantly reduce the computational cost associated with required KP-ABE crypto operations.

Using symmetric-key crypto. In *K2C* the actual content of data is encrypted using a symmetric-key encryption scheme and *only* the corresponding symmetric key is encrypted by KP-ABE scheme. By default our framework uses AES (Advanced Encryption Standard) [39] for symmetric encryption with the default key length of 128 bits. Similarly, *AB-SIGN* signature scheme is performed on fixed-length digest of data. Our framework, by default, uses SHA-1 [1] as the digest hash function. SHA-1 generates 160-bit message digest of data.

The following figures show the overhead of our encryption and signature schemes on top of underlying symmetric-key encryption and hashing schemes. In our experiments numerical attributes are of size 10 bits.

Figure 2.4 shows how the cost of *K2C* encryption and decryption relates to the user's access level and hierarchy level of the data object. In KP-ABE the encryption time is *only* a function of number of attributes, which linearly increases as the object level increases. As a result, *K2C* encryption cost linearly increases as the hierarchy level of the object increases, but it is independent of the user's access level.

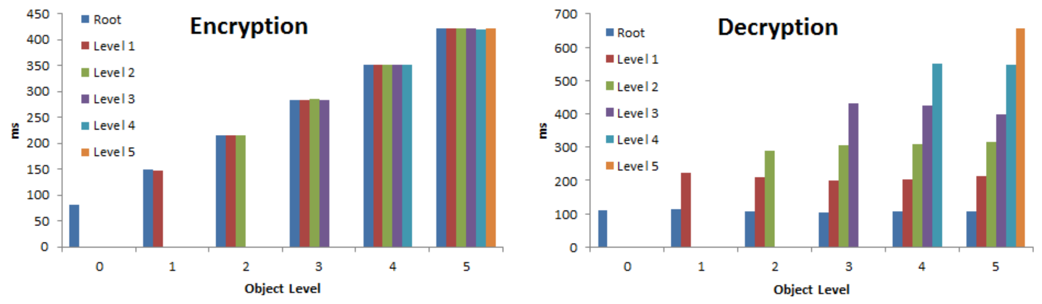

Figure 2.4: Overhead of *K2C* encryption and decryption of objects in different hierarchy levels for users with different access levels.

By contrast, as figure 2.4 shows, decryption time is just a function of user's access level. That is because in KP-ABE, decryption time is a function of complexity of access structure that linearly increases as user's access level increases. Decryption time is independent of hierarchy level of the encrypted object.

In *K2C*, as Figure 2.5 illustrates, signature cost is independent of the hierarchy level of data objects; it only depends on access level of the user. This is because our *AB-SIGN* signature scheme is based on KP-ABE derive operation which its complexity linearly increases as the complexity of the access structures increases.

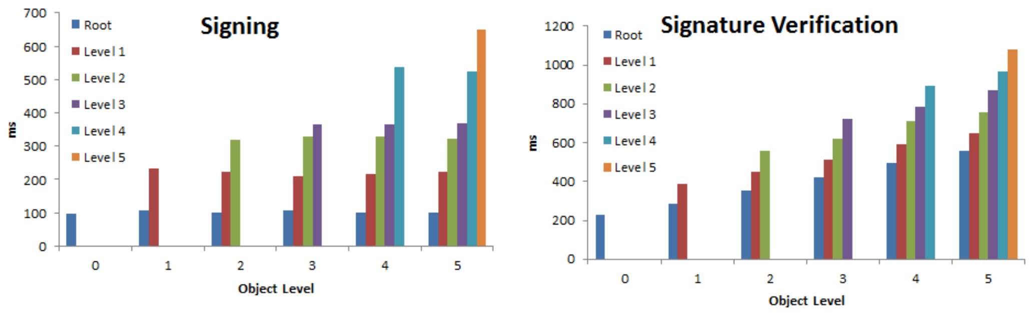

Figure 2.5: Overhead of *K2C* signing and signature verification on signed objects in different hierarchy levels for users with different access levels.

In *AB-SIGN* scheme, each signature verification operation requires KP-ABE encryption and decryption, therefore its computational cost depends on the user's access level as well as the hierarchy level of data object. Figure 2.5 shows how the time required for signature verification increases linearly as the access level of user decrease and the hierarchy level of data object increases.

Figure 2.6 shows the combined overhead cost of read and write operations in *K2C*. To perform a *K2C* write operation, a user needs to encrypt and sign the data objects. The portion of cost below the white indicator is related to encryption and the rest is the cost associated with signature. A *K2C* read operation includes cost of decryption and signature verification. The value below and above the white indicator shows the overhead cost for decryption and signature verification, respectively.

2.6 Related Work

There are two general key management approaches which are used in the existing cryptographic file systems: 1) classic access control list (e.g., [53, 60]) requires maintaining a key list along with each file. This approach supports fine-granularity but is not scalable. 2) grouping files and assigning the same access key to each group (e.g., [62]). This approach is more scalable but provides coarse-grained access control. This trade-off makes these solutions unsuitable for clouds where we need a fine-grained and scalable

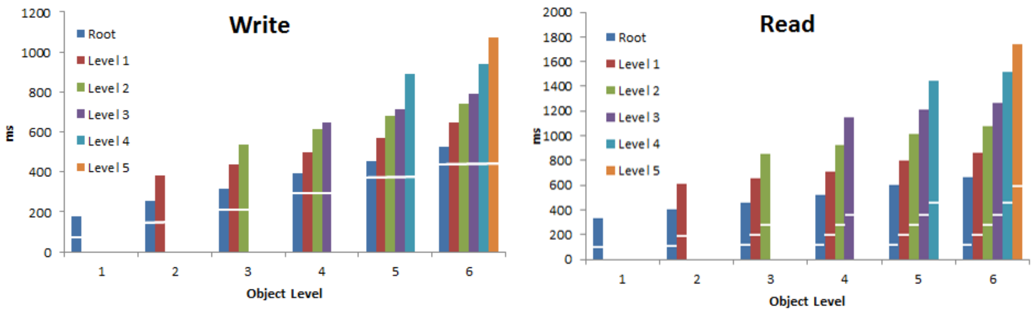

Figure 2.6: Overhead of *K2C* write and read operations on objects in different hierarchy levels for users with different access levels.

access control mechanism. In [88], Shucheng Yu et al. introduced a novel approach which addresses this trade-off by proposing a fine-grained and scalable access control protocol. Their solution uses *lazy re-encryption* to statistically reduce the number of re-encryptions required after access revocation. They use *proxy re-encryption* (PRE) [28] to off-load the task of re-encryption to cloud servers. In our solution, we adopt *lazy revocation* to eliminate these re-encryptions. In [86], Xiong et al. introduce a protocol for securing end-to-end content distribution when delivery services are involved.

Lazy revocation was first introduced in Cephues [45] to eliminate re-encryption required for each revocation at the cost of slightly lowered security. Lazy revocation, which is widely being used in recent cryptographic file systems [62, 78, 81], requires a key-updating scheme to support key regression. Key-updating schemes are studied and formalized in [20]. In [53], Grolimund et al. introduced Cryptree which can support access hierarchies and lazy revocation simultaneously. However, due to the explicit and physical dependency of these links, file system operations – especially revocations – require updating large number of these cryptographic links. For example, the revocation of write privilege requires updating $O(n)$ keys, where n is the number of data objects contained in that folder and its sub-folders. Therefore, revocation of write access for a folder containing many files is relatively slow as all the links that connect to the contained sub-folders and files need to be updated.

Moreover, in Cryptree, since key derivation requires traversing cryptographic links, key derivation time is a function of distance of data objects to the folder that the user has access to. Therefore, users with access to high-level folders (e.g. root user) have slower read access. For a specific user read access time depends on the location of the data object, but intuitively we expect the read access time to be independent of the location of the data object. Another limitation of this approach is that Cryptree does not support the delegation of administrative rights and assumes that granting and revoking access rights are done by a single administrator, an assumption which is usually unrealistic in the context of Cloud Storage, as we expect non-centralized administration of data. In this paper we introduced a scalable key updating scheme for hierarchies which addresses these shortcomings and enables us to build a cryptographic access control supporting lazy revocation.

2.7 Summary

We presented a novel key-updating scheme that can be used to enhance the scalability and performance of cryptographic cloud storages by adopting lazy revocation. We also designed a new digital signature scheme that enables cloud providers to ensure that requests are submitted by authorized end-users, without learning their identities. Using our key-updating and signature schemes, we developed, implemented, and evaluated a scalable cryptographic access control protocol for hierarchically organized data.

Chapter 3

Client-side Service Mashup

In this chapter we present the design and implementation of our browser-independent integration framework called *OMOS*(Open Mashup OS).*OMOS* framework is designed for dynamic, seamless and secure integration of cloud services. In Web 2.0 terminology applications that seamlessly combine contents from multiple heterogeneous data sources into one integrated browser environment are called Mashups. The hallmark of these applications is to facilitate dynamic information sharing and analysis, thereby creating a more integrated and convenient experience for end-users. Currently, developers use ad-hoc and often insecure techniques to develop mashups. As mashups evolve into portals designed to offer convenient access to cloud resources and critical domains, concerns to protect users' personal information and trade secrets become important, thereby motivating the need for strong security guarantees. We develop a security architecture that provides high assurance on the mutual authentication, data confidentiality, and message integrity of mashup applications, thereby is suitable for integration of cloud services. In the next chapter, we show how using this framework we can develop secure and privacy-preserving identity management protocols.

This chapter is organized as follows. In Section 3.1 we introduce the security challenges that exist in development of mashups and explain our design goals. The architecture of *OMOS* framework is presented in Section 3.2. In Section 3.3 we explain implementation details of *OMOS* framework. The security analysis is in Section 3.5. Related work is explained in Section 3.6. We give the conclusions in Section 3.7.

3.1 Introduction

Mashup applications are emerging as a Web 2.0 technology to seamlessly combine contents from multiple heterogeneous data sources; their overall goal is to create a more integrated and convenient experience for end users. For example, http://mapdango.com is a mashup application that integrates Google Maps data with relevant information from WeatherBug, Flickr, Eventful, etc. By entering a location, the user is presented with an integrated view of the weather of the location, events happening in surrounding area, photos that others took in the area, and so on. There are two main types of architectures for mashup applications, namely, server-side and client-side architectures. As the name indicates, server-side mashups integrate data from different sources at the server-side, and return the aggregated page to the client. For example, Facebook mashup APIs are mainly based on server-side integration [44].

However, the main drawback of server-side mashups is the requirement of complete trust on the mashup server by the client. Typically, the client needs to delegate authorization to the mashup server to act on its behalf.

In comparison, a client-side architecture, as illustrated in Figure 3.1, enables consumers and service providers to communicate within a browser, thus reduces the amount of trust that one has to place on untrusted third-party integrator. OpenSocial provides a client-side mashup API [72]. Throughout this paper, we focus on client-side mashup architecture, as emerging mashup applications using AJAX techniques hold the promise of the next technical wave of the future [18]. AJAX, short for asynchronous JavaScript with XML, is a technique that allows a Web page to retrieve contents from the Web server and update the page asynchronously using JavaScript. AJAX mashups are able to present a rich user interface and interactive experience with multiple data sources with minimal transmission delays.

3.1.1 Security Challenges

Client-side mashup architectures allow information mashup to happen within the client's browser through the use of JavaScript. A mashup application should be able to access

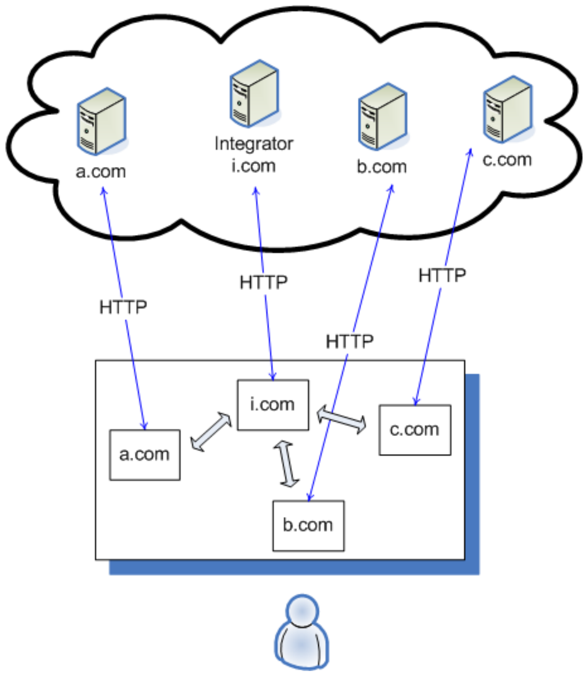

Figure 3.1: Client-side mashup architecture. The rectangle represents the browser on the client's local computer where contents from heterogeneous data sources such as a.com and b.com are mashed up.

and integrate contents from different sources. In general, there is a trade-off between the security and functionality in today's mashup applications. In order to achieve higher security guarantees, a source should not be allowed to access contents of another domain. The `frame` or `iframe` element in the current browsers realizes this separation by forbidding one frame from accessing another frame with a different source domain. However, frame environments make it awkward for cross-frame communication and thus information integration in mashups.

To mitigate issues caused due to lack of communication method between iframes, HTML5 introduces a new API called `postMessage` [22]. The `postMessage` API provides a way for verification of sender's and receiver's origin, however that remains the responsibility of the developer to do the actual verifications. For the reasons that we discuss in the next section, in practice the required verifications are not trivial and therefore often developers skip the required verifications which leads to various vulnerabilities. Moreover, According to Facebook [22] only about half of their users' browsers support `postMessage`, therefore for compatibility reasons they were using some ad-hoc non-secure technique for inter-frame communication. (trade-off for functionality and compatibility over security.)

To address these problems, several client-side mashup architectures have recently been proposed, including MashupOS [83], Subspace [58] and SMash [63]. The main goal of these solutions is two-fold: to isolate content from different sources in sandboxes, such as frames, and to achieve secure frame-to-frame communication.

SMash [63] uses the concepts in publish-subscribe systems and creates an event hub abstraction that allows the mashup integrator to securely coordinate and manage content and information sharing from multiple domains. The mashup integrator (i.e., the event hub) is assumed to be trusted by all the web services. The event hub implements the access policies that governs the communication among domains.

MashupOS [83] introduces a sophisticated abstraction that enables web components from different domains to securely communicate. OMash [38], inspired by MashupOS, tries to simplify the abstraction and remove the reliance on same origin policy (explained in Section 3.1.4). However, implementing these abstractions requires adding new elements to the HTML standard and major changes in the browsers to support them.

Subspace [58] suggests an efficient techniques for `www.mashup.com` to use a JavaScript Web service from `webservice.com` by sandboxing the Web service in a frame that is originated from a *throwaway* sub-domain (e.g. `webservice.mashup.com`) and communicating with it by shortening document.domain to a common suffix and passing a JavaScript object that can be used for secure communication. The main drawback of this approach is that, due to the same origin policy, the Web service running from `webservice.mashup.com`, cannot use `XMLHTTPRequest` to communicate with its resources on backend site (`webservice.com`) and this restriction limits the use of AJAX Web services.

However, none of these solutions provide a flexible and secure point-to-point inter-domain communication mechanism that can be used in today's browsers without relying on developers to ensure the security of communication.

3.1.2 Relevance of OMOS in the postMessage era

HTML5 introduces `postMessage` [22] API as a standard component of web. This new API greatly facilitates client-side inter-domain communication. Previously this method was only available in a few bowers such as Opera, but HTML5 guarantees that this API will be consistently available in all browsers in near future. Using `postMessage` API, a sender who has a reference to another iframe can forward a message to that iframe. This method has an optional input parameter that receives the expected domain name of the target iframe and ensures that the message will be delivered only if the domain name of the target iframe is equal to the specified domain. Note that due to Same-Origin-Policy (SOP) there is no other direct way for the sender to check the domain name of the target iframe. On the receiver side, upon arrival of each message the receiver can query the origin of the message to learn the domain name of its sender. Since this API is directly supported by the browser, it provides a very efficient way for client-side communication thus will result in increased adoption of the client-side integration paradigm.

Although `postMessage` API can greatly improve the quality of client-side integration and also provides all information required for ensuring authenticity and confidentially of communication, but in practice it cannot directly satisfy all the requirements of non-trivial mashups in terms of functionality and security. This is mainly due to the fact that `postMessage` API only provides a low-level one-way mechanism for physical transfer of data from one iframe to another and completely relies on the developers to to use it securely. However, in practice interactions of mashup components are two-way and in the form of asynchronous request-reply. This makes the process of tracking messages and ensuring that they will be properly routed and transfered to the right domain complex and tedious.

A recent study [54] on usage of `postMessage` shows that due to these limitations and complexities developers tend to skip the required verification and this leads to various vulnerabilities. As part of their research, they investigated two prominent users of the `postMessage` primitive, the Facebook Connect protocol and the Google Friend Connect protocol, and discovered many security vulnerabilities caused by direct usage of this low-level communication API. In their evaluation, they observed that developers, belonging to the same organization and sometimes of the same application, used the primitives safely in some places while using them unsafely in others. Their research shows that the reason that developers fail to follow the recommended practice and required sanity checks is the complex cross-domain interactions involving fine-grained origins. For example, in a dynamic mashup environment requests can come from any domain and even one domain may have multiple request sessions. In these cases, validating the origin of sender, enforcing access control on each request and making sure that the corresponding response will be sent back to the correct party is non-trivial and very tedious. When highly talented and trained developers at Google and Facebook fail to use the API securely, we can expect to see more of these types of vulnerabilities as the usage of this API becomes more and more widespread.

To address these problems, the OMOS framework introduces a communication stack that provides a set of very flexible high-level APIs that abstracts away all the underlying complexity of client-side communication, and performs all the required sanity checks consistently according to best practices. This not only addresses the security concerns associated with direct and inconsistent usage of this API, but also lets the developers concentrate on implementation of the actual service rather than dealing with performance and security details of communication. Besides ensuring security of communication, the OMOS framework addresses security problems such as frame Phishing, access control and user authentication.

3.1.3 Design Goals

To close the security gap that we explained in the previous section, we designed and implemented *OMOS* framework considering the following design goals:

- To be compatible with all major browsers without any change or extension to the browsers.

- To provide a powerful abstraction that is flexible and easy to understand and use by mashup developers.

- To guarantee mutual authentication, data confidentiality, and message integrity in mashup applications. (Defined in Section 3.1.4.)

The novel features of our approach are as follows. First, we present key-based protocol for secure asynchronous point-to-pint inter-origin communication. Second, we separate communication layer from access control layer, therefore the framework can be used using different access control mechanism. Third, we present a layered communication abstraction for inter-frame communication fashioned after the networking stacks that is both powerful to use and easy to understand. Additionally, the framework does not require any browser change, so it is a good candidate for secure development of today's mashup applications.

The following techniques, enable us to achieve all of our design goals. Our key-based protocol satisfies the security requirements and prevents attackers from phishing, forging, tampering, and eavesdropping on cross-domain communications. Since we do not require new HTML elements, *OMOS* is compatible with current browsers. The layered abstraction hides implementation details from mashup developers and the API allows anyone to extend and improve any part of the mashup framework. OMOS' communication API and component-based development also make the development of complex AJAX applications much easier. (Reusable components are called mashlets in *OMOS*, Section 3.1.4.)

An additional advantage gained by using our techniques is that the mashup integrator (i.e., mashup site) need no longer be trusted by all the content providers (i.e., web services). This is possible because the frames from different web services are able to directly and securely communicate within the user's browser. Therefore, with *OMOS* it is possible to create new types of mashup applications that may involve and integrate sensitive and personal data without fully trusting the mashup integrator. For example, banking, shopping, and financial planning applications contain important personal information that users want to have high assurance on the controlled sharing of data. Allowing different domains to communicate in a secure fashion minimizes the potential risks of information exposure due to corrupted websites such as compromised mashup integrators, and untrusted contents from other data sources. We have implemented and evaluated the performance of the *OMOS* framework on five types of browsers. These initial experiments show that the communication channels are able to deliver high throughput without affecting the end user's browsing experience.

3.1.4 Definitions

We define mashlets, gadgets and mashup applications. A *mashlet* is recursively defined as a HTML frame hosting JavaScript service that contains zero or more mashlets. The root mashlet is always visible and is usually called a mashup container. Every mashlet is controlled by and loads contents from its originating domain. Conceptually, mashlets are analogous to processes or daemons in the operating system, binary components (e.g COM/DCOM, DLL) in component-based architectures or web service providers in service oriented architectures (SOA). A *gadget* is a mashlet that is visible in the browser. A *mashup application* is a gadget that integrates data received from other mashlets [1].

Two most important aspects of mashup applications are interaction and security. Interaction refers to the ability of a mashlet to interact with its siblings, children, and parent mashlets. Security requires that a mashlet should not be able to access private information, such as DOM elements, events, memory, and cookies, of any other mashlet that is running under a different domain. In particular, a mashlet should not be able to listen to the communication between two other mashlets running under different domains. We call this requirement *data confidentiality*. In today's browsers, the same origin policy (SOP) [79] is designed to protect data confidentiality of domains against each other; in other words, SOP prevents documents or programs from one origin to access or alter documents loaded from another origin. SOP restrictions on JavaScript that govern the access to inline frames (`iframes`) [2] forbid JavaScript in one mashlet including the root mashlet to read or modify the contents in another mashlet.

[1]This definition concentrates on client-side mashups

[2] Frames that can be inserted within a block of texts.

However, SOP is restrictive and rigid for mashup applications in general. Mashlets from different domains are isolated and cannot communicate or interact unless specifically allowed. Most of existing mashup applications circumvent this restriction either by creating server-side mashups, which is a less flexible approach, or by allowing complete access from other domains. Recently, researchers also demonstrated the vulnerabilities associated with carelessly attempting finer-grained origins [57]. In HTML5 a new API, `postMessage` [22], has been introduced that facilitates inter-domain communication between iframes.

Mutual authentication is another important security requirement in cross-domain mashlet communication. We define *mutual authentication* in mashup applications as the requirement that two mashlets that are communicating with each other must be able to verify each other's domain name.

Mashup applications should also satisfy the *message integrity* requirement that means that any tampering of the messages between two mashlets should be detected. *OMOS* satisfies the three requirements of data confidentiality, mutual authentication, and message integrity, by leveraging the security restrictions available in current browsers and by developing a lightweight key establishment protocol.

3.2 Architecture of *OMOS*

In this section, we first give an overview of *OMOS*, and present its layered communication stack for inter-mashlet communication. Finally, we present some important implementation details of our technique.

3.2.1 Components

Our goal is to support secure, asynchronous, inter-mashlet communication in browser environments. Much of our design in *OMOS* is lead by existing inter-process communications in networking, e.g., TCP. That is, we model the cross-domain frame-to-frame interactions (i.e. a frame communicating with another frame of a different domain) in a manner similar to cross-domain process-to-process interactions in networking paradigm.

We develop a layered communication model for the purpose of cross-domain frame-to-frame communications that can be easily extended.

The *OMOS* framework can be viewed as a container for mashlets that manages their construction, destruction and resources, also provides them with services such as communication, persistence, user interface, user authentication and pub-sub messaging. Services that *OMOS* provides to mashlets are analogous to services that operating systems provide to desktop applications through well-defined APIs. *OMOS* runs entirely in the browser, requires no browser plug-in, and supports all main stream browsers, including Firefox, Internet Explore, Safari, Opera, and Chrome. Figure 3.2 illustrates how mashlets using *OMOS* interact.

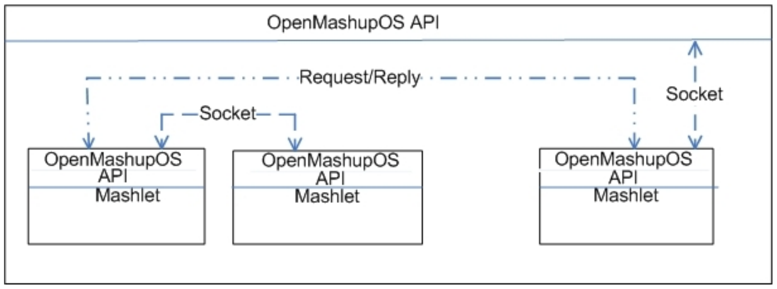

Figure 3.2: Interactions between mashlets in *OMOS* framework. Each mashlet connects to the integrator using a socket connection that *OMOS* uses to provide services to mashlets.

OMOS uses iframes to implement mashlets. For each mashlet loaded from a distinct domain, existing SOP restriction guarantees the confidentiality and isolation of mashlets. Also *OMOS* provides mashlets with a flexible, reliable, asynchronous and secure communication service that guarantees data confidentiality, data integrity, and mutual authentication using a layered communication stack.

3.2.2 Layered Communication Stack

Our *OMOS* architecture abstracts the mashlet communication and provides mashup developers with a powerful and flexible API. We borrow the concepts in networking to design a communication stack in *OMOS*. The administrative communications between mashlet and the parent mashlet (i.e., integrator) are done using a *socket connection*.

Most of the *OMOS* service calls through JavaScript APIs lead to a communication through this socket connection. As a result, we are able to support modularity and transparency. Complex implementation details are hidden from the outside. For example, the request to get the DOM address of a specific domain name (or principal) is invisible to mashup developers. Figure 3.3 depicts the communication layers in *OMOS* architecture, namely from bottom to top, Datalink layer, Mashup Datagram Protocol (MDP) layer, and Mashup Hypertext Transfer Protocol (MHTTP) layer.

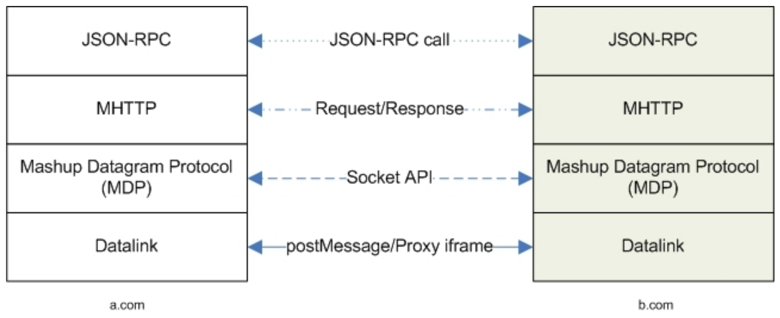

Figure 3.3: Communication stack in *OMOS*. The arrows and their texts are the communication methods for the layers. Note that all the communications between two mashlets take place within the end user's browser.

At the Datalink layer, communications are realized in a direct frame-to-frame fashion, which needs to be compliant with restrictions imposed by browsers. For example, the size of data to be transferred is limited depending on the type of the browser and the communication method, and DOM location of an iframe (e.g., parent.frame[3]) is used for addressing. We further discuss the Datalink layer services and implementation techniques in Section 3.3.1.

The purpose of Mashup Datagram Protocol (MDP) layer is to abstract the Datalink layer details. MDP provides the *logical client-side communication* between two mashlets, in such a way that from a mashlet's perspective, it is directly sending arbitrary sized data to another mashlet. Yet, in reality, the data may be fragmented, defragmented, and re-ordered which are all handled in the lower Datalink layer. Mashup applications use the logical communications provided by the MDP layer to send data to each other, without worrying about the implementation details of browser types, restrictions, etc. In MDP layer domain names and port numbers are being used for addressing. *OMOS* exposes services provided by this layer using socket APIs that is very similar to Java socket API for conventional TCP/IP communication. *OMOS* uses socket connection for administrative communication between mashlets and their parents. During the bootstrapping process, when a mashlet is first loaded, it gets the communication parameters from the segment identifier of its URL provided by the integrator, i.e., parent mashlet. Then the mashlet creates a socket connection to the integrator service on port zero (dedicated for this purpose). Through this bootstrapping process, the integrator establishes connections to all the mashlets that it contains. The integrator uses these connections to provide the services that the mashlets need, e.g., finding the DOM location of a specific mashlet, changing the width and height of their iframes, or resolving domain names to frame address, etc.

Mashup HyperText Transfer Protocol (MHTTP) is the top layer in the communication stack of *OMOS*. MHTTP provides stateless *request* and *reply* types of communication and abstracts all the details of socket programming. It is very common for service consumers that need to send a request to a service provider and get the corresponding response. It is easy for service providers to define the interface for these types of services with MHTTP.

We use JSON-RPC protocol on top of the MHTTP layer [61]. JSON-RPC is a simple lightweight remote procedure call protocol that is very efficient in AJAX applications [18]. This layer makes it easy to use existing JavaScript services. Instead of directly injecting JavaScript code, service consumer includes the service in a sandbox mashlet and hosts the mashlet in a safe throwaway subdomain. Then the service consumer uses JSON-RPC to call the service and retrieves the result without giving the script full access to resources available in the main domain.

3.3 Implementation Details

In this section, we describe some important implementation details of *OMOS*. Our descriptions of our communication stack are bottom-up, starting from the Datalink layer.

3.3.1 Datalink Layer

Datalink is the layer that does the actual transfer of data from one frame to another. *OMOS* currently uses iframe proxy or postMessage (if available) for cross-domain communication between frames. Other communication mechanisms can be implemented and easily plugged into the framework. In Opera and some especial configuration of other browsers, frame navigation is restricted that prevents two mashlets in different frame hierarchies from communicating directly. In this case, if the integrator is not trusted then the communication fails and *OMOS* will prompt the user to use a browser with permissive navigation policy; otherwise, the data link layer or the integrator mediates and routes the data link packets to the destination.

For inter-frame communication, if `postMessage` API is not available, *OMOS* fails to *iframe proxy* techniques to do inter-frame communication. *OMOS*uses a simple key exchange protocol and the fact that browsers enforce a *write-only policy* on URL field of iframes to ensure confidentiality and authenticity of communication. If the `postMessage` API is present (e.g. in HTML5 or Opera), we set `targetOrigin` properly to make sure the message will be delivered only to the correct domain, also check the `sourceOrigin` property of the received messages to ensure that they are coming from an authentic sender. In the next section we provide more details on iframe proxy communication and our key exchange protocol.

iFrame Proxy and Key Establishment Protocol

Browsers enforce a *write-only policy* on URL field of iframes, which means that a frame can *write* to the URL field of a frame with a different origin domain, but *not read*. The URL field of a frame can only be read by the frame itself or a frame of the same origin. Therefore, in *OMOS*, if iframe A originated from `a.com` wants to pass some data to iframe B from `b.com`, iframe A creates an internal temporary hidden iframe that points to a proxy page that is hosted on b.com and sets the fragment identifier to carry data (for example, http://b.com/proxy.html#data). As part of its OnLoad event, the proxy page reads the data from its URL and delivers that to iframe B. The iframe proxy gets removed afterward. This method has the following benefits over the approach that is used in [63]; it is event driven and does not require polling, therefore eliminates the delay between each poll and improves the performance by eliminating unnecessary timers. With this solution, we eliminate the click sound problem that IE has in SMash [3].

Although this event-based communication mechanism through iframe proxy has been documented elsewhere [40], [37], it is not known previously how to achieve mutual authentication in this communication method. When frame A writes http://b.com/proxy.html#data as the URL in the iframe proxy, A can make sure that frame B can get the data only if its domain is b.com (because of SOP); however when frame B receives data, there is no direct way to find out the origin of the received data. We develop a key establishment protocol in *OMOS* that is used by two frames to initiate a shared secret key. By leveraging the write-only property of frame URL, the key establishment protocol elegantly allows the two frames to verify each other's domain name (e.g., that iframe[A]'s domain is a.com and iframe[B]'s domain is b.com).

OMOS key establishment protocol is as follows. Let say frame[1] from a.com and frame[2] from b.com want to exchange a shared secret key. Frame[1] generates the secret key SK_1 and passes the key to frame[2] using a proxy from b.com. Since frame[2] can get the key only if it is originated from b.com, frame[2] can prove that its origin is b.com by responding back with SK_1. However, b.com still needs to verify that the origin of frame[1] is a.com. To do so, it generates a new secret key SK_2 and passes it along with SK_1 using a.com's proxy then frame[2] can prove that its origin is a.com by responding with SK_2. At this point only a.com and b.com know SK_2 so they can use it as a shared secret for the rest of communication. Note that key establishment happens during three-way handshake in MDP layer that is described in the next section. Using this protocol, *OMOS* framework can provide mutual authentication capability in

[3] A click sound is usually made in IE when a frame is redirected, which can be distracting if it occurs too frequently as the frames URL gets repeatedly updated for the data transfer purpose.

inter-mashlet communication.

Figure 3.4 illustrates this key establishment protocol. Data fields shown on the arrows between the two frames represent Datalink packets, which encapsulate MDP packets. SK_1 and SK_2 are session secrets chosen by frame[1] and frame[2], respectively for each communication session. EID is an identifier needed by Datalink layer for addressing destination object. Each frame also creates a serial number in each Datalink packet.

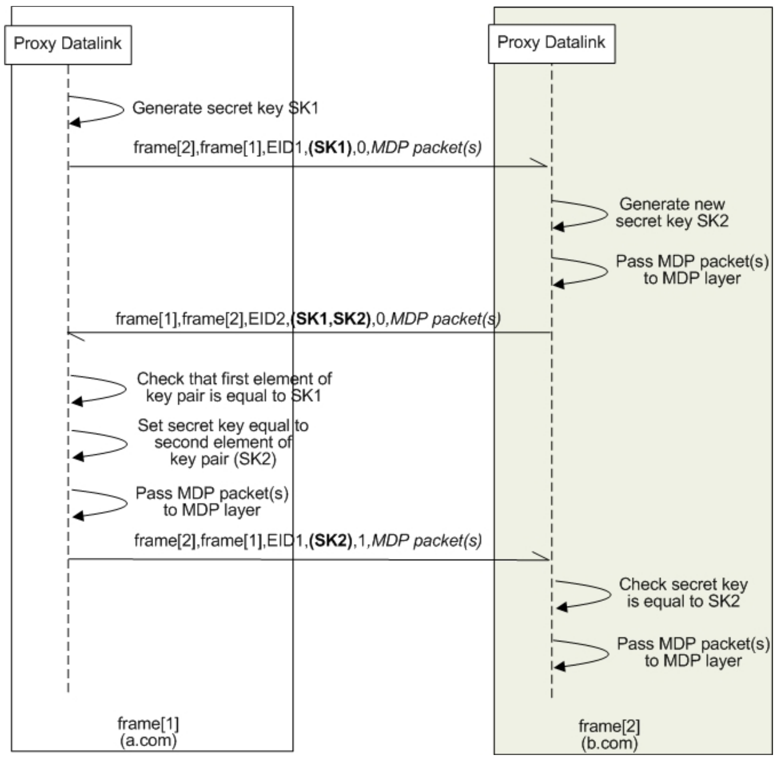
Proxy Datalink
Proxy Datalink
Generate secret key SK1
frame[2],frame[1],EID1,(SK1),0,MDP packet(s)
Generate new secret key SK2
Pass MDP packet(s) to MDP layer
frame[1],frame[2],EID2,(SK1,SK2),0,MDP packet(s)
Check that first element of key pair is equal to SK1
Set secret key equal to second element of key pair (SK2)
Pass MDP packet(s) to MDP layer
frame[2],frame[1],EID1,(SK2),1,MDP packet(s)
Check secret key is equal to SK2
Pass MDP packet(s) to MDP layer
frame[1] (a.com)
frame[2] (b.com)

Figure 3.4: Key establishment protocol between two mashlets, frame[1] from a.com and frame[2] from b.com, through Datalink layer.

Other services of Datalink layer

Besides physical tranfer of data, the Datalink layer provides services like reordering, (de)fragmentation, and (un)piggybacking to enable efficient transfer of arbitrarily big data objects or frequent events/small objects. We explain them as follows.

- **Fragmentation:** Each browser defines Maximum Transfer Unit (MTU) size that specifies the maximum amount of data that can be transfered without interrupting responsiveness of user interface. Generally, `postMessage` API does not impose any hard limit but transferring large chunk of data will affect end-user's experience. In proxy-based communication MUT is set to the maximum size that is allowed for the URL field. When the size of a MDP packet is larger than MTU, the packet should be fragmented to smaller chucks and then sent to the other end, which is called fragmentation. On the other side, the receiver's DataLink layer assembles these fragments and sends the resulting MDP packet up to MDP layer, which is called defragmentation. This service enables transfer of arbitrarily large data objects without interrupting responsiveness of user interface.

- **Reordering:** We observe that in some cases, depending on how event handling is implemented in the browser, packets sent using iframe proxy and also `postMessage` sometimes arrive out of order. Reordering ensures that MDP packets are delivered to MDP layer in the order that they are sent by the sender.

- **Piggybacking:** Piggybacking essentially refers to a lazy-send approach for transferring small data objects. *OMOS* needs to create a new iframe proxy for every data transfer between two iframes. When the sender has frequent small sized data objects, it is more efficient to collect them and send them together using only one iframe proxy, instead of sending them in multiple iframe proxies. To do so, *OMOS* automatically detects this case and keeps the small data objects in a queue and piggyback them on an single iframe proxy. This service dramatically improves the event rate. Similarly in the case of `postMessage`, piggybacking reduces the number of generated events and reduces the overhead of communication.

3.3.2 MDP Layer

In *OMOS*, MDP (Mashup Datagram Protocol) is similar to transport layer protocols in TCP/IP (or UDP/TCP). However, note that all of the frame-to-frame communications occurred in *OMOS* take place in the end user's browser on the user's *local* *machine*, as *OMOS* supports client-side mashups. The inter-frame messages are represented by the thick arrows in Figure 3.1. An MDP communication has three phases: 1) Connection establishment (three-way handshake) 2) Communication (transferring actual data) 3) Disconnection (upon requests of one of the peers, closing the connection and releasing the resources). Figure 3.5 illustrates these three phases. Note that all mashlet-to-mashlet communications are asynchronous. Applications can communicate at the MDP layer using *OMOS* socket APIs. The APIs are asynchronous meaning that actions are executed in non-blocking scheme, allowing the main program flow to continue processing. Programs pass callback functions to handle events. Figure 3.5 shows a usual MDP communication scenario;

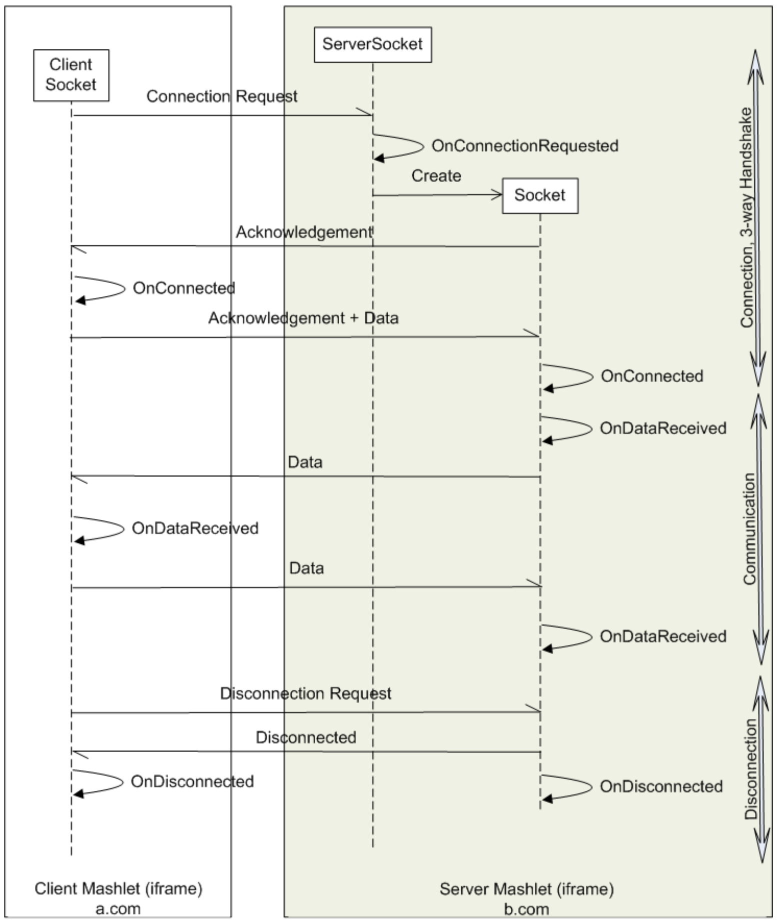

Figure 3.5: Connection establishment (three-way handshake), Communication, and Disconnection are three phases of a typical MDP communication session.

The following code snippets illustrate how *OMOS* socket APIs can be used for providing and consuming data/service. In code snippet 3.1, the service provider starts a server socket on port 1111. When a client connect, the server socket sends the current time to the client.

Listing 3.1: Server Socket Setup

```
var serverSocket = OMOS.ServerSocket(1111);
var ssCallback =
{
   onConnectionRequested: function(socket)
   {
     /*Define the server socket to handle
        onDataReceived, onTimeout, and onError events */
      var sCallback = ...
     // Set the callback object to server-side socket endpoint
      socket.setCallback(sCallback);
      var currentTime = new Date();
     //Send data to client that is connected to this socket
      socket.send(currentTime.getTime());
   },
   onError: function(exp) {/*handle exception */}
}
serverSocket.accept(ssCallback);
```

The code snippet 3.2 illustrates how a service consumer can connect to a server socket on port 1111 of a mashlet hosted by domain `time.example.com`. Once the client receives the current time from server, after presenting it to the user, closes the connection.

Listing 3.2: Client Socket Setup

```
var sCallback =
{
  onConnected:function() {alert("Connected to server");},
  onDisconnected:function() {alert("Disconnected.");},
  onDataReceived:function(socket,data) {
                    alert("Server's time is "+data);
                    socket.disconnect();},
  onTimeout:function() { /*handle timeout  */ },
    onError:function(exp){ /*handle exception */ },
    timeout:1000
}
var socket = OMOS.socket("time.example.com",1111, sCallback);
```

3.3.3 MHTTP Layer

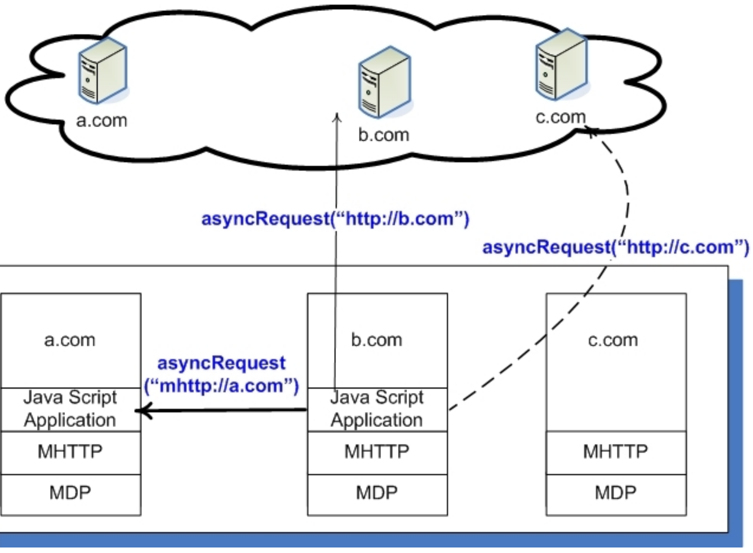

Figure 3.6: Illustration of the flexibility of asyncRequest method in *OMOS* that can be used to realize three types of requests from b.com: same-domain mashlet-to-server communication (solid thin line), cross-domain mashlet-to-server communication (dash line to server at c.com), and mashlet-to-mashlet communication (solid thick line to mashlet at a.com).

OMOS provides the main functionality of the MHTTP layer through the versatile `asyncRequest` method that abstracts the same-domain and cross-domain HTTP calls to servers as well as the mashlet-to-mashlet communication. The latter happens inside the browser on the client's local machine. The implementation of the `asyncRequest` method is built on the existing `XMLHttpRequest` API in JavaScript. Currently, `XMLHttpRequest` only handles same domain mashlet-to-server interaction. Our `asyncRequest` realizes cross-domain requests by coupling `XMLHttpRequest` with our mashlet-to-mashlet communication mechanism (described in previous sections). Thereby, we are able to provide a nice and clean interface for all three types of calls, which are shown in Figure 3.6.

The code snippet 3.3 shows how we can use *OMOS* API to make a `MHTTP` call:

Listing 3.3: Client Socket Setup

```
var callback =
{
  onDataReceived: function(response){ /*consume response */ },
  onTimeout: function(){ /*handle timeout   */ },
  onError: function(exception){ /*handle exception */ },
  timeout: 1000
}
OMOS.asyncRequest('POST',
                "mhttp:5555//socialnetwork.com/service",
                callback,
                JsonRpcRequest
                );
```

3.4 Performance Evaluation of iFrame Proxy

The goal of the experiments is to test the performance of *OMOS* iframe proxy communication method in various browsers. We ran experiments on a machine with the following configurations. Intel Core 2 CPU, 980 MHz, 1.99 GB RAM, Microsoft Windows XP 2002 SP2, Firefox v2.0.0.14, Internet Explorer v7.0.5730.13, Opera v9.27, and Apple Safari v3.1.1. The values reported are the averaged results over five runs.

Figure 3.7 shows the throughput as the size of messages increases in FireFox, IE, and Safari. FireFox and Safari have similar performance in terms of throughput as they both can achieve around 430 KB/s of transfer rate. Recall that MDP layer can handle arbitrarily large data objects. The underlying Datalink layer handles the URL limitation by fragmentation and defragmentation. For IE, the throughput is much lower and can achieve the transfer rate of 50 KB/s. The slowdown in IE is due to the URL limit (2KB) imposed by IE, as there is overhead in the Datalink layer to fragment and defragment large messages into small packets that can be fit into 2KB URL. The mashlets communicate through iframe proxies described in our protocol. In general, larger message sizes give higher throughput for all three types of browsers.

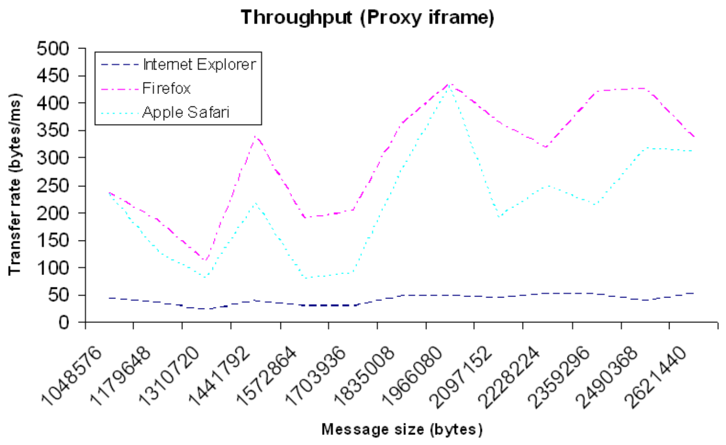

Figure 3.7: The figure shows the throughput between two mashlets with iframe proxies in FireFox, IE, and Safari. X-axis is the size of MDP packets.

Opera gives high throughput, due to the native support of inter-frame messaging (i.e., postMessage [22]), it is shown in a separate graph in Figure 3.8. Figure 3.8 shows that Opera gives throughput as high as 2500 KB/s with message sizes around 2MB. However, the performance then degrades as the message sizes increase. The transfer rate eventually drops to zero as the message size reaches around 2.6MB. The root cause of this poor performance of Opera with larger message sizes is currently not clear to us. From the throughput results, 2 MB seems to be the optimal message size.

Even though using the larger message sizes (i.e., frame URL) for transferring data leads to higher throughput, we observed that using very large message sizes leads to less responsive user interface and thus affects the surfing experience of the end user. Based on our experiences, the maximum message size should be around 100 KB to ensure responsive browser interface. Therefore, there is a trade-off between performance and usability. IE's URL limit affects the rate of information transferred and significantly slows down the data transfer. In comparison, for all the other three browsers, the frame URL can be very large (> 2MB). *OMOS* is able to find the suitable size of frame URL automatically.

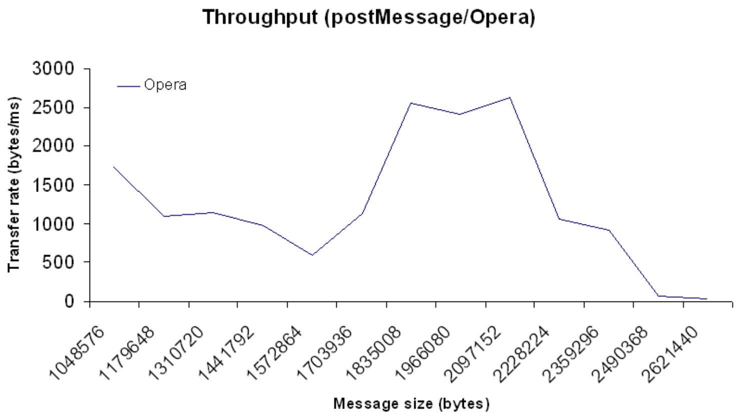

Figure 3.8: The figure shows the throughput between two mashlets with PostMessage in Opera. X-axis is the size of MDP packets.

3.5 Security Analysis

We analysis the security of *OMOS* from three aspects: data confidentiality, message integrity, and mutual authentication. We describe how frame phishing can be easily prevented in our framework.

Data Confidentiality. *OMOS* satisfies the data confidentiality requirement of inter-mashlet communication by leveraging the browser's same origin policy. SOP protects the message that is being transfered through `postMessage` API or iframe proxy from being read by any domain other than the target domain. In the case of proxy-based communication, the sender passes the message through the URL of a proxy iframe hosted by the domain of the intended receiver. Since SOP enforces write-only restriction on the URL field of iframes, the URL can be read only by proxy's domain, an therefore no man-in-the-middle can read the message. In the case of `postMessage`, `targetOrigin` input ensures that the message can be read by the receiver iframe only if it is hosted by the intended domain.

Message Integrity. In proxy-based communication message integrity is realized by utilizing the browser's restriction on partial change of URLs and the shared key between two frames. To modify any data carried on the URL, a mashlet needs to know the secret key, otherwise the packet is rejected and dropped at the destination. Thus, an unauthorized mashlet is unable to tamper with inter-frame messages. SOP guarantees that a message being sent by `postMessage` cannot be modified during transmission.

Mutual Authentication. In proxy-based communication, our key establishment protocol in Figure 3.4 guarantees that the mutual authentication between two frames, say frame[1] from a.com and frame[2] from b.com, is achieved in *OMOS*. Frame[2]'s origin is successfully authenticated, if and only if it sends back the secret key SK_1 sent by frame[1] through b.com's proxy. Similarly, frame[1] proves its origin by sending SK_2 back to frame[2]. The confidentiality of communication ensures that frame[1] and frame[2] are the only two mashlets that know SK_2. If `postMessage` is being used for the communication, *OMOS* sets the `targetOrigin` parameters properly to make sure that the message will be delivered to the right target and also upon arrival of each message *OMOS* checks its origin and make sure that the corresponding response will be delivered back to the same sender.

Detecting Frame Phishing. Frame phishing refers to where a malicious frame in a mashup can change which frame is loaded in another part of the mashup [63]. For example, an attacker's frame can change bank.com's frame to point to attacker.com, which may mislead the end user into disclosing sensitive information such as password or banking data. The mashlet's parent in *OMOS* can conveniently detect this type of frame phishing. A regular mashlet has an on-going socket session with its parent for administrative commands. In a normal scenario, disconnection of this session is initiated by mashlet or its parent and this session should be closed before mashlet gets unloaded. Therefore, if attacker.com redirects bank.com to a malicious frame, since the administrative session is still alive, bank.com mashlet, as part of its onunload even handler, will send a phishing attack notification to its parent. Therefore, parent mashlet can take the appropriate action and notify the end user of the threat by prompting an alert window, for example.

Access Control. *OMOS* framework separates communication and access control mechanism. Therefore, different access control techniques can be used to control communications between mashlets. For example, a policy enforcement mashlet can govern communication of different mashlets similar to central even hub in SMash, or in a distributed fashion, each mashlet can control access to its services using a *dynamic whitelisting* technique. In the next chapter we present more details on access control and authentication protocols which are designed and developed based on *OMOS* framework.

3.6 Related Work

The authors of MashupOS recognized that existing browser has a limited all-or-nothing trust model and protection abstractions suitable only for a single-domain system [83]. They proposed new abstractions for the content types and trust relationships in the current browser environments. In MashupOS, new native HTML tags are introduced to HTML page. These tags can be added and removed dynamically using JavaScript, so mashups with dynamic layout are possible. To demonstrate the feasibility, the authors have implemented their abstraction using browser plug-in for IE in such a way that browser at compile time converts them to standard HTML tags and simulates their functionality. The main difference between MashupOS and *OMOS* is that MashupOS provides a modified browser, whereas we create library supports that applications can use within current browsers. In Subspace [58], JavaScript web services are placed into iframes that are originated from "throw-away" subdomains of mashup integrator. This approach is not flexible in general, as web services need to run under the subdomain of the integrator, and cannot directly perform XMLHtmlRequest calls to their backends.

Keukelaere *et al.* developed SMash that is a secure component communication model for cross-domain mashups called SMash [63]. In SMash, all of the communications are through the mashup integrator, which is also called hub. The hub mediates and coordinates all the communications via tunnel frames among the participating frames. The hub also enforces access policies. It prevents frames from eavesdropping on or tampering the others' communication channels. SMash inter-frame communication is supported through a tunnel frame pointing to the integrator's domain that each frame needs to create in order to communicate with the integrator.

In comparison to SMash where a tunnel frame exists in every mashlet, we create an iframe for every round of communication and send the information encoded in the fragment identifier during an onLoad event. Therefore, unlike SMash, we do not need a polling mechanism and the communication in *OMOS* is event-driven. Polling creates negative impacts on the performance of single-threaded browser. We support mutual authentication in our inter-mashlet communication that prevents an attacker from frame spoofing. In our *OMOS*, cross-domain frames can communicate directly without the participation of the mashup integrator. Therefore, the trust assumption put on the mashup integrator can be relaxed.

Recently, a secure postMessage method is proposed by Barth, Jackson, and Mitchell [22]. They have proposed a protocol to fix an authentication vulnerability in several (polling-based) inter-frame communication protocols including SMash, and Windows Live communication protocol [68]. The communication protocol used in *OMOS* dose not have this issue, as is explained in Section 3.5.

Cross-site request forgery (XSRF), which is also known as the confused deputy attack against a Web browser [17], is a malicious attack again websites by exploiting browser vulnerabilities. In a XSRF attack, a malicious website can launch an iframe to make requests on behalf of the user to another website with which the user's authenticated session is still valid. For example, the request may be to transfer funds from the user's bank or to change the user's Gmail configuration. A secure browser *OP browser* that prevents and detects XSRF was presented by [52]. Simple alternatives are for websites to set a short expiration period on authenticated sessions, and to educate users to close authenticated sessions upon finishing.

Singh and Lee presented a browser design inspired by μ-kernel based OS [65] that allows flexible and finer customization. The main design difference between Singh-Lee browser and OP browser is that OP browser is process-based whereas Singh-Lee browser is within the same address space that makes it possible for the browser to provide memory isolation for browser components. As with other mashup solutions (SMash and MashupOS), *OMOS* depends on the security of browser to correctly operate. Therefore, a secure browser such as OP would be complementary to our techniques in realizing web security.

Recently, Hanna *et al.* studied usage of `postMessage` API, which was recently introduced in HTML5, in a set of real-world applications [54]. Their results show that many applications, for various reasons, do not use this API securely. They propose the economy of *liabilities principle* for designing future abstractions.

3.7 Summary

We presented our design and implementation of a secure and efficient communication framework *OMOS* for mashup applications. *OMOS*provides a flexible and powerful set of API for client-side inter-domain communication and abstracts away the implementation details and handles all security related challenges such as authenticity and integrity of communication, access control and frame phishing protection.

Chapter 4

User-centric Identity Management System

This chapter addresses the identity management problems in integrated cloud-based mashup environments. We present Web2ID, a new identity management framework tailored for mashup applications that integrate multiple cloud services. Web2ID leverages a secure mashup framework and enables transfer of credentials between cloud service providers and consumers. We also describe a new relay framework in which communication between two service providers is mediated by a relay agent within the mashup. We show that Web2ID is privacy-preserving and prevents service providers from learning a user's surfing habits.

This chapter is organized as follows. In Section 4.1 we provide a short introduction to some identity management related challenges that exist in today's mashup environments and summarize our contributions in addressing these issues. We provide the background information on mashup-related concepts in Section 4.2. We give our threat model in Section 4.3. Our basic Web2ID protocol is in Section 4.4. Extensions of Web2ID are presented in 4.5 and 4.6. Our implementation is presented in Section 4.7. We apply Web2ID to the identity management problem in webtop applications in Section 4.8. Related work is described and compared in Section 4.9. We give the conclusions at the end of this chapter.

4.1 Introduction

Mashup applications integrate information from multiple autonomous data sources within the Web browser for a seamless browsing experience. For example, iGoogle allows users to create personal pages containing "gadgets" from multiple Web domains, such as NYTimes, Weather.com and Google Maps. Despite their popularity, mashups are still not in widespread use for sensitive cloud applications. Such mashup applications currently require user authentication to prevent unauthorized access to sensitive information. For example, financial mashups such as mint.com and yodlee.com allow users to view a summary of their financial activities by accessing back-end services, such as banks and credit card companies. However, users have to reveal their credentials to these mashup service providers in order to delegate them access to their financial data.

For historical reasons the Internet lacks unified provisions for identifying who communicates with whom [76]. Therefore, currently service providers use ad-hoc solutions to identify and authenticate their users. These ah-hoc solutions require the user to create a new identity for each service provider. This leads to multiple isolated identities for each user; a problem which is particularly troublesome in mashup applications as they need to authenticate and cores-reference users across multiple service providers in order to share users' protected resources with their consent.

Most existing identity management protocols for the Web, including OpenID [2], use a unique URL to represent the identity of a principal. The advantage of using a URL as opposed to a name or email address is that a URL is tangible, clickable, user-friendly, and can contain information that facilitates the authentication process. This URL is called the principal's *identity URL* . The static page that is located at identity URL is called the *identity page*. The server that hosts the identity page is called the *identity host*. Therefore, during authentication, users claim ownership of their identity URL and proves their claims to a service provider by following the corresponding authentication protocol. However, all these authentication protocols require a trusted third party, called *Identity Provider (IdP)*, to validate the user's claim. Users first create an account with IdP and use the identity page to delegate the authentication of their identity URL to that IdP. But this feature may compromise user privacy as the IdP can learn the surfing habits of the user.

In this work, user privacy refers to the protection of a user's transaction history. Better privacy refers to the separation of providers (service providers or identity providers) in a way so that they do not directly communicate to each other regarding to the users and their activities. This notion of privacy follows the one used by Goodrich, Tamassia, and Yao in a federated identity management or single sign-on (SSO) model [50]. Unlike [50], our solution is decentralized and does not require any centralized server.

The main advantage of Web2ID is that it uses public-key cryptography to enable users to prove the ownership of their identity URL without relying on third parties. Authentication in *Web2ID* and PGP [3, 85] are similar as they both eliminate the need for a centralized identity provider, and thus support decentralized trust management. However, the main difference is that *Web2ID* is specifically designed for modern web 2.0 mashup application. In Web2ID, the identity of each user is represented by a URL whereas in PGP the public keys directly represent the identity of users. This feature makes Web2ID more user friendly as URLs are easier to remember. Moreover, Web2ID provides powerful and flexible APIs that can be easily imported and integrated into existing modern Mashup applications.

Web2ID relies on client-side components that run within browser and are able to securely communicate with each other. In this paper we use the term *mashlet*, which we introduced in [90], to refer to these components. We provide more details on mashlets in Section 4.2. In *Web2ID*, users are represented by a mashlet hosted at their identity URL, in much the same way that service providers are represented on the client-side by their mashlets. We call the mashlet that is hosted at the identity URL an *identity mashlet*. That is, in *Web2ID*, the identity page is a mashlet, (i.e., it includes JavaScript libraries required for communication) which provides authentication services.

During the authentication protocol, users first presents their identity credentials to their identity mashlet. In turn, the identity mashlet acts on behalf of the user and interacts with other mashlets to prove that the user owns the identity that corresponds to its URL. The identity mashlet enables other desirable features including authorization delegation and attribute exchange.

We have three main technical contributions in this paper.

1. We design and implement a new identity-management framework – Web2ID – for

sensitive client-side mashup applications. Web2ID supports identity authentication on the Internet without any centralized trusted party. In addition, user's privacy (in terms of service history) is protected because identity providers and service providers do not communicate directly about the user's requests.

We also describe how attribute exchange and the delegation of authorization are done in Web2ID.

2. We implement an in-browser public-key cryptosystem in JavaScript for Web2ID cryptographic operations can be completed solely in the browser. Our library is general-purpose and is useful beyond the specific identity management problem studied.

3. We generalize Web2ID to the identity management in Webtop applications, where a browser provides a desktop-like environment for SaaS applications such as office applications and data management systems. We describe our open-source Webtop application and how to integrate Web2ID with it. The source code for Webtop and cryptographic libraries are available on SourceForge ([92]).

In our Web2ID architecture, we define a new communication structure which we call *mashlet relay.* The mashlet relay is a mashlet that passes information between two mashlets belonging to different domains. Thus, it enables *indirect* cross-domain communication. For identity management, mashlet relay enables a service provider to send a query to another provider without revealing its identity. The mashlet-relay framework protects user privacy in mashup environments because service providers that host user data are unable to learn how users consume their data. This feature is especially important when users wants to provide their identity attributes (certified by a trusted party) to another service provider.

Our framework is implemented as a JavaScript library without browser modifications or specialized plugins to operate. It is fully portable across browsers and execution platforms. We illustrate the portability of our framework by incorporating it with several popular browsers, including Firefox, Opera, Apple Safari, IE and Google Chrome. Moreover, we avoid using HTTP redirections for communication; consequently, our protocol is compatible with modern Ajax-based Web applications.

4.2 Background and Definitions

In this section, we present background material on mashup frameworks, discuss the problems addressed by our identity management protocol, and conclude with a description of an existing relatively popular identity management protocol (OpenID).

4.2.1 Mashups and Mashlets

Mashup applications aggregate content from a number of providers and display them within Web browsers. Such applications can be designed either as *server-side* mashups or *client-side* mashups. In server-side mashups, a proxy (called the mashup server) aggregates content from multiple sources. The Web browser loads the mashup application by visiting a URL corresponding to the proxy. For example, Facebook applications use the RESTful API provided by Facebook to query a user's social information and aggregate it with other data. In contrast, client-side mashups directly aggregate content within the Web browser. Several frameworks have recently been proposed to support safe yet expressive client-side mashups [22, 42, 58, 63, 83, 90]. In this paper, we restrict our attention to client-side mashup applications.

The client-side components of a mashup application are called *mashlets*. Mashlets represent the service provider that is hosting them in the client side and run in the browser with the privileges given to their hosts. To be concrete, a mashlet is simply a HTML page which loads to an iframe and contains some JavaScript code that enables it to communicate with other mashlets in the page. A *mashup application* is a Web application that aggregates a number of mashlets, possibly from different sources on the Web. We also use the term *mashlet container* to refer to the mashup application.

A number of recently-proposed frameworks allow mashlets to securely communicate with other mashlets executing in a mashup application [22, 58, 63, 83, 90]. A secure inter-mashlet communication protocol is one that guarantees mutual authentication, data confidentiality, and message integrity. *Mutual authentication* in inter-mashlet communication means that two mashlets that communicate with each other must be able to verify each other's domain name. *Message integrity* requires that any attempt to tamper with the messages exchanged between two mashlets should be detected/prevented. *Data confidentiality* means a mashlet should not be able to listen to the communication between two other mashlets running under different domains.

For concreteness, the rest of this paper describes mashups and mashlets in the context of OMOS (Open Mashup OS) [71, 90], a secure client-side mashup framework that we developed in prior work. However, the concepts developed in this paper are applicable to any client-side mashup framework that provides the above properties.

4.2.2 Identity Management in Mashup Applications

In the following discussion, we consider three problems in identity management and discuss how the *Web2ID* protocol and our mashup relay framework address each of these problems. Where applicable, we also discuss why existing techniques fail to address the problem.

User Authentication. When sensitive Web applications, such as those for banking, investment and tax services, are integrated into a mashup environment, it is highly desirable to use an authentication protocol that provides single sign-on (SSO). SSO enables these service providers (i.e., the bank or the investment company) to authenticate the user without requiring her to prove her identity separately to each provider. The goal of an authentication protocol in a mashup environment is for the user to prove the ownership of an identity to a service provider without revealing any information that can be misused by a malicious service provider to impersonate the user.

Most existing identity management protocols for the Web, including OpenID [2], use a unique URL to represent the identity of a principal. The advantage of using a URL as opposed to a name or email address is that a URL is tangible, clickable, user-friendly, and can contain information that facilitates the authentication process. This URL is called the principal's *identity URL*. The static page that is located at identity URL is called the *identity page*. The server that hosts the identity page is called the *identity host.* Therefore, during authentication, the user claims ownership of an identity URL and proves her claim to a service provider by following the corresponding authentication protocol. However, all these authentication protocols require a trusted third party, called the *Identity Provider (IdP)*, to validate the user's claim. Users first create an account with IdP and use the identity page to delegate the authentication of their identity URL to that IdP. But this violates user privacy because the IdP can learn the surfing habits of the user.

Web2ID uses public-key cryptography to enable users to prove ownership of their identity URL without relying on third parties. In *Web2ID*, users are represented by a mashlet hosted at their identity URL, in much the same way that service providers are represented on the client-side by their mashlets. We call the mashlet that is hosted at the identity URL an *identity mashlet.* That is, in *Web2ID*, the identity page is a mashlet, i.e., it includes JavaScript libraries required for communication and providing authentication services.

During the authentication protocol, the user first presents her identity credentials to her identity mashlet. In turn, the identity mashlet acts on behalf of the user and interacts with other mashlets to prove that the user owns the identity that corresponds to its URL. The identity mashlet enables other desirable features including authorization delegation and attribute exchange. We define both problems below.

Attribute Exchange. An important feature supported by most identity management frameworks is that of *attribute exchange*, in which one service provider requests a user's identity attributes (e.g., her age) or preferences from another service provider. Attribute exchange is especially important for mashup applications, in which interaction between mashlets is the norm. We refer to a service provider that requests user's attributes as an *attribute requester* and the service provider that stores user attributes and settings as an *attribute provider* (also called a wallet [74]). An attribute provider may optionally certify user attributes (e.g., for attribute-based authorization) or simply send non-certified values (e.g., for providing settings and preferences).

An identity attribute exchange protocol should ideally accommodate three privacy requirements:

- **Requirement 1:** An attribute provider should share a user's attributes only upon explicit consent from the user.
- **Requirement 2:** An attribute requester should be able to query a user's attributes without necessarily knowing the identity of the user.
- **Requirement 3:** The protocol should be able to anonymize an attribute requester to prevent an attribute provider from learning identity of the requester (and thereby, the user's Web surfing habits).

Designing a browser-based protocol that can satisfy all these requirements is challenging. Existing browser-based attribute exchange protocols use a series of HTTP redirections to keep users in the loop to acquire their consent without revealing their identity to the requester (Requirement 1). However, in such protocols, the requester must send a callback URL to the provider; as a result, HTTP redirection-based communication discloses the identity of the requesters, thereby violating Requirement 3. State of the art techniques to remove the need for communication between the attribute requester and provider use sophisticated cryptographic techniques (e.g., idemix [32]). However, these solutions are not currently suitable for practical use in browser-based protocols [74]. Web2ID uses our proposed mashlet relay framework to anonymize the attribute requester.

Authorization Delegation. Web application mashlets included in a mashup typically access resources hosted at other domains. In this context, the mashlet that accesses resources is typically called *Consumer*, while the domain that hosts the resource is called *Service Provider*. Consumers should not be able to access a user's protected resources unless the user grants them the required access permission.

An *authorization delegation protocol* allows a user to delegate permissions to a consumer to access her resources hosted at a service provider. For example, a user may be able to delegate permissions needed to access her files on a photo-sharing website (the service provider) to a website that provides photo editing utilities (the consumer). An authorization delegation protocol should be privacy-preserving in that it must not reveal the user's identity. In the example above, for instance, the user may wish to grant the photo editing service read access to her photos hosted on the photo sharing website without revealing her identity to the photo editing service.

4.2.3 OpenID Protocol

In this section, we outline the OpenID protocol, which is relatively a popular identity management solution on the Web, and explain its shortcomings for the next generation of Web applications. The main purpose of this section is to give some background to the reader that is not familiar with OpenID. We have deliberately omitted details, and encourage interested readers to consult the OpenID specification.

OpenID was originally designed to enable bloggers to leave comment on others' blog posts without requiring them to create new new accounts in each blog providers. OpenID requires the users to define their identities on a trusted third party Web site that serves as the IdP. Service providers can verify a user's claim for ownership of an identity url (e.g., Alice.me) by contacting the identity provider using the OpenID protocol. According to the OpenID standard, the protocol is executed via a series of communication sessions between the the IdP and the SP that happen via a sequence of HTTP redirections, i.e., the SP page redirects the browser automatically to the IdP, which then redirects the browser back to the SP page with the user's credentials.

Description of the OpenID protocol. Suppose that Alice wants to adopt the URL Alice.me as her OpenID identity. Alice needs to first select an identity provider that she trusts (e.g IdP.com) and then register herself, say as Alice@IDP.com. Alice must then embed information about her trusted IdP along with her identity into the page that is located at Alice.me (her identity page). This means that the owner of the URL Alice.me should be able to prove that she owns identity Alice@IDP.com on IdP.com. Figure 4.1 depicts the operation of the OpenID protocol. It shows how a service provider can verify the claim of ownership of an identity URL. As this figure shows, the protocol proceeds in eight steps:

1. The user claims the identity URL Alice.me.

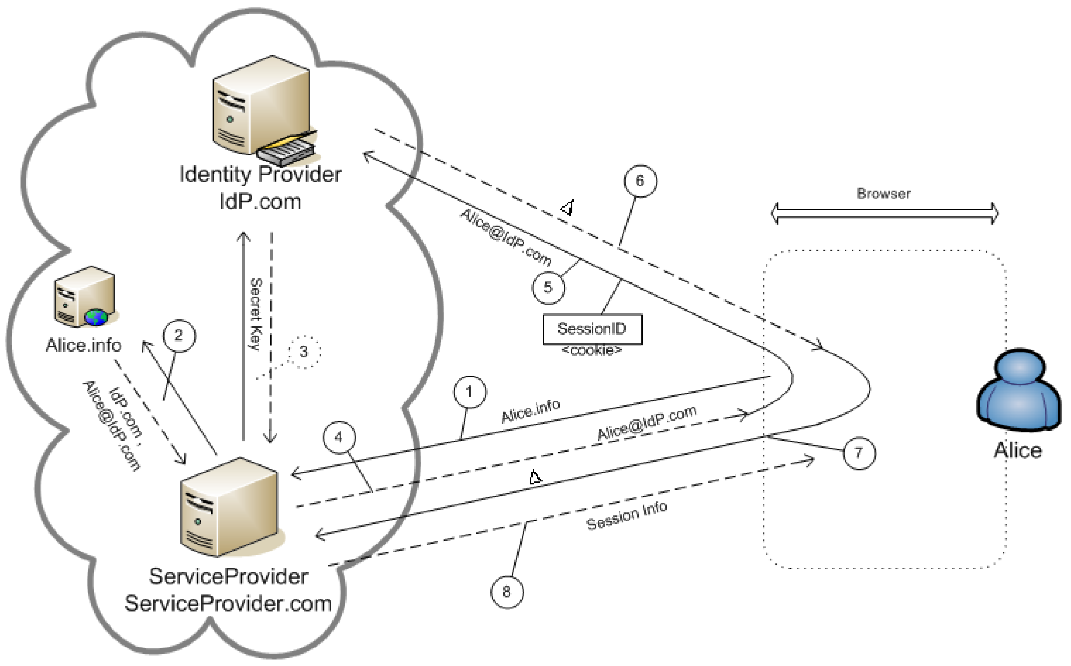

Figure 4.1: This figure illustrates how using OpenID a service provider can use authenticate a user.

2. From the identity page located at the claimed URL, the service provider retrieves information about IdP (service end point URLs) and delegate ID (e.g. Alice@IdP.com).

3. If the service provider and the identity provider have not already established a shared secret key, they exchange a secret key using the Deffie-Hellman protocol.

4. Service provider sends an HTTP redirect response back to the user's browser (agent).

5. The user's browser redirects to the identity provider. If the user has an authenticated session with identity provider then this request also carries a cookie containing the session info .

6. Identity provider first checks the session information to see if the user is authenticated and has a valid session ID.

   If the user is not already authenticated then IdP returns a login page asking the user to login. The user enters and submits her login credentials for Alice@IdP.com and identity provider authenticates the user (for clarity, this step is not shown in the figure).

After ensuring that the user's identity is Alice@IdP.com, the IdP generates an assertion (Δ) using a secret key shared with the service provider stating that the user owns the identity Alice@IdP.com, and sends a redirect HTTP response to the browser.

7. After receiving the HTTP redirect response, the browser makes an HTTP call back to the service provider passing the assertion.

8. Service provider validates the assertion using the corresponding shared secret key and if it is valid, service provider can make sure that user's claim of ownership of the identity Alice.me is valid.

Shortcomings of the OpenID protocol. Although a well-accepted standard in the domain of traditional Web applications, we argue that OpenID is ill-suited for the domain of Ajax-based Web-2.0 applications for several reasons.

First, the redirection-based approach to IdP/SP communication dictated by the OpenID standard fails with Ajax-based Web-2.0 applications. Redirection requires the browser to unload the currently-loaded Web application and load a new page. As Web applications move from stateless, static entities to dynamic, stateful ones, unloading and reloading a page is no longer practical. For example, consider a Web-based desktop application with several windows open for Word-processing, Email-processing and financial applications using spreadsheets. Simply unloading such a Web-based application will result in loss of unsaved data. Creating Web applications that support checkpointing of state is complex and is unlikely to be adopted by Web application developers.

Second, the OpenID protocol is vulnerable when applied in the Web-2.0 domain. For instance, OpenID uses the Diffie-Hellman key-exchange protocol to exchange secret keys between an SP and a consumer who wish to share a user's identity. However, the Diffie-Hellman protocol is known to be vulnerable to the Man-in-the-Middle attack.

Although the OpenID specification suggests that this attack can be avoided using SSL (https), this approach is rigid and may not be practical for several Web-2.0 applications.

Third, implementing SSO using OpenID requires storing session information in a cookie that will be sent to the identity provider. However, as described in prior work [21], doing so renders the protocol vulnerable to cross-site request forgery attacks.

In addition to the above imitations of OpenID, several other smaller limitations preclude OpenID from being adapted easily to environments that mashup information from Ajax-based Web-2.0 applications. For example, implementing the OpenID protocol requires both the SP and the IdP to maintain state information. This complicates the implementation of the protocol and also makes it unscalable to large deployments. In addition, a trusted third party (the IdP) is required for authentication. This may not always be a practical assumption, and even if a third party is available, users with privacy concerns may be unwilling to share their surfing habits with the third party.

These concerns motivate the need for a new identity management protocol for Ajax-based Web-2.0 applications.

4.3 Our Threat Model

In Web2ID, there are several types of players: user, service provider, service consumer, attribute provider, and outside attacker (or stranger). Each player has a different degree of trustworthiness, as we explain next.

Users may be malicious. As is standard with AJAX-based applications, some messages of the Web2ID protocol are exchanged on the client-side, within the user's browser via inter-mashlet communication. Because the user has complete control over the browser, a malicious user may alter the client-side component of the Web2ID protocol, for example, by forging the identity of another user or providing forged identity attributes to an attribute requester. Consequently, for transactions in which the user must not be trusted, the correctness and integrity of the Web2ID protocol must not rely on the client-side portion of the protocol executing correctly. Web2ID uses cryptographic techniques to ensure the integrity of data that passes through the client.

Service providers may be malicious. When a service provider authenticates a user, it must receive certain information that enables it to ensure the authenticity of the user. A malicious service provider may misuse this information to impersonate the user to a second service provider using a *relay attack*. For example, a malicious service provider attacker.com that authenticates Alice may use her credentials to impersonate her to another service provider honest.com. In this attack, attacker.com tries to log into honest.com claiming the ownership of Alice's identity URL (e.g., alice.me). When honest.com challenges attacker.com, it relays that challenge to Alice when she tries to prove her identity to attacker.com. In turn, attacker.com uses this information to convince honest.com of Alice's identity.

Service consumers may be malicious. In the authorization delegation protocol, a malicious consumer may try to convince a service provider to give it access to a user's protected resources without possessing appropriate authorization (i.e., explicit consent from the user). In the case where users wish to protect their privacy from the consumer, a malicious consumer may try to learn the user's identity during the course of authorization.

Attribute providers may be malicious. We refer to a service provider that requests user's attributes as an *attribute requester* and the service provider that stores user attributes and settings as an *attribute provider* (also called a wallet [74]). An attribute provider may optionally certify user attributes (e.g., for attribute-based authorization) or simply send non-certified values (e.g., for providing settings and preferences).

In an attribute exchange, a malicious attribute provider may try to violate a user's privacy by learning the identity of requesters that try to obtain the user's attributes. As a result, the attribute provider may learn the user's surfing habits. Similarly, a malicious attribute requester may also try to learn the identity or attributes of the user without user's agreement.

Man-in-the-Middle (MitM) attacks. Based on their capabilities, man-in-the-middle attackers (MitM) [56] can be either active or passive. A passive MitM attacker only listens to the conversation between two parties in the protocol. The goal of a passive attacker is to obtain information that can be used to impersonate the users, get unauthorized access to their private resources or violate their privacy. In contrast, an active attacker can also modify the content of conversation. An active MitM may try to change the result of an authentication or authorization check by modifying or replaying data transmitted in the protocol. MitMs can also be classified based upon their location in the network. Client-side MitMs involve a malicious mashlet that tries to spoof mashlet-to-mashlet communication in the protocol. A network MitM spoofs network communication, such as those between a mashlet and its server, or between two servers. In the Web2ID protocol, we assume that the point-to-point network communications are safe against active MitM attacks, which can be guaranteed by using secure lower level protocols like SSL. Finally, malicious mashlets may also try to subvert the protocol by launching frame phishing attacks against the user [63].

4.4 Basic Web2ID Protocol

The basic Web2ID protocol enables users to prove their identity to a service provider website without the use of a trusted third party. This protocol enables users to independently prove their identities and prevent any third party from learning their surfing habits. We achieve this goal using public-key cryptographic primitives in a manner akin to public-key client authentication in SSH (RFC 4252 [4]).

In Sections 4.5 and 4.6, we extend our basic Web2ID protocol to support attribute exchange and the delegation of authorization, respectively.

Suppose that a principal P (e.g., Alice) wishes to adopt an identity I (e.g., an identity URL, such as alice.me) and prove her ownership of that URL to a service provider SP.com. There are two main operations in the basic Web2ID protocol: *identity adoption* and *user authentication*, as described in the following.

Identity Adoption. To adopt an identity URL I, say alice.me, Alice first hosts an identity mashlet at this URL. The identity mashlet is a component that is trusted by Alice and represents her within a mashup application. To configure her identity mashlet, Alice must navigate to her identity mashlet using a browser. When the identity mashlet loads for the first time, it detects that it is not configured, and generates a public/private-key pair ($Pu(I)$,$Pr(I)$). The public key is embedded within the identity mashlet, while the private key must be stored safely by Alice.

User Authentication. When a user such as Alice attempts to authenticate herself with a service provider, she claims the ownership of an identity URL, such as alice.me. In turn, the service provider sends a session token encrypted under the public key associated with the identity URL alice.me. When the user then sends requests to access resources, she must prove ownership of the session token corresponding to her claimed identity. Figure 4.2 illustrates how service provider SP.com assigns a session token to the user who claims the ownership of identity alice.me. As Figure 4.2 illustrates, authentication happens in seven steps, as described below:

1. The user claims to own an identity I. For instance, this identity could be an identity URL alice.me. This claim can be communicated to the mashlet of the service provider SP.com. For example, the user may enter the URL in a form provided by SP.com.

2. The service provider's mashlet sends the claimed ID I to the service provider and, if not already loaded, loads the identity mashlet located at the claimed identity URL (i.e. alice.me).

3. The service provider extracts the public key $Pu(I)$ and the type/version of the corresponding public-key encryption algorithm Alg from the claimed identity page.

4. The service provider first generates a session token χ, and encrypts χ and the domain name of its mashlet SP.com with the public key $Pu(I)$. It then sends the result $\Delta = E_{Pu(I)}(\chi, \text{SP.com})$ back to the service provider's mashlet as response. Note that the domain name of the service provider must be included in Δ to protect users against relay attacks by malicious service provider (see Section 4.3).

5. The service provider's mashlet sends $\Delta = E_{Pu(I)}(\chi, \text{SP.com})$ to the identity mashlet for decryption.

6. If the identity mashlet does not already have the private key $Pr(I)$, it asks the user to provide her login credentials. Using the user's login credentials identity

mashlet computes the private key. For example, the user can load the encrypted value of her private key from a USB memory stick and provide a passphrase that can be used by the mashlet to compute the private key. Alternatively, the user may enter the private key directly by swiping a smart card that contains her private key.

Once the identity mashlet has the private key and user permits the authentication, the identity mashlet decrypts Δ and verifies that the domain name of the service provider (SP.com) matches the domain name in the token.

7. The identity mashlet sends the computed session token χ back to the service provider mashlet.

In our implementation, inter-mashlet communication is facilitated by the OMOS framework [90] which provides mutual authentication, and confidentiality and integrity guarantees for the data exchanged between two mashlets. *This ensures that a malicious mashlet in the mashup application will not be able to compromise communication between the identity mashlet and the service provider's mashlet* (so the value χ will not be available to an eavesdropper).

Upon the completion of the above protocol, the service provider can verify that the value of the session token received from the identity I is valid. This proves the user's claim of ownership of I to SP.com. Existing identity management protocols prove the possession of the session token by including it with each request, and are therefore vulnerable to session hijacking via MitM attacks. Our implementation of Web2ID uses a MAC (message authentication code) to prove possession of the session token.[1] In this approach, the MAC value of each `XMLHttpRequest` request is computed using the session token and is included in every request. The service provider serves a request only if the included MAC value is correct. Note that during the above protocol does not require the service provider to keep any protocol-specific state, thereby ensuring a stateless implementation of the web application at the service provider. In addition, the

[1]To do so, we ported the necessary cryptographic functions HMAC-SHA1 and HMAC-SHA256 (RFC2104 [64], RFC3174 [41]) into the OMOS framework.

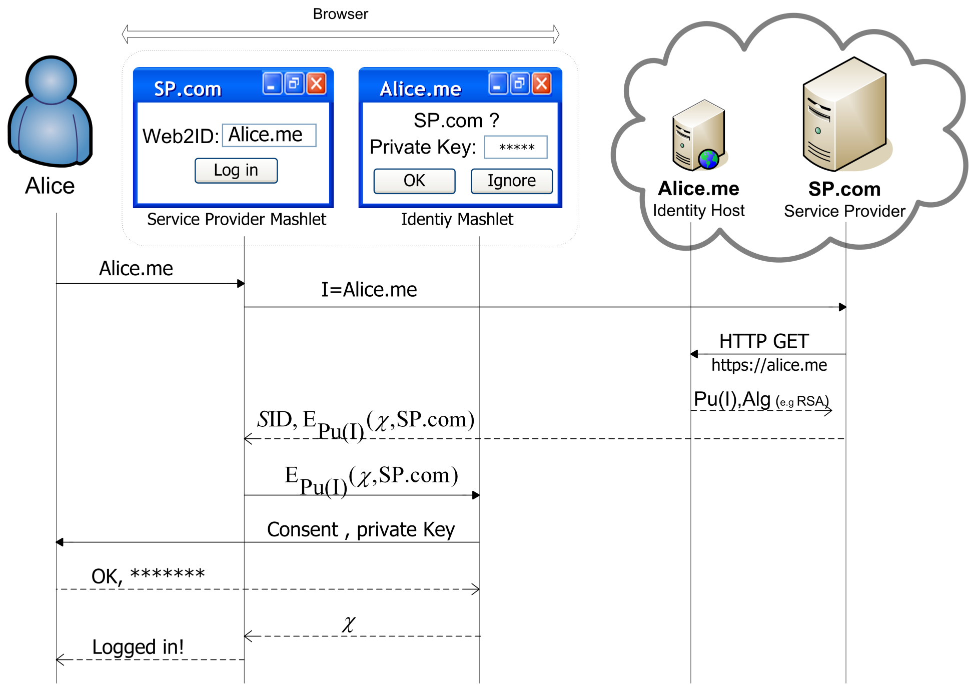

Figure 4.2: An identity mashlet represents the user within the application. The user can prove ownership of the identity mashlet by proving the possession of the private key that corresponds to the public key located at URL of the identity mashlet.

user's credentials are never transmitted over the network; instead such communication happens on the client-side, where communication is secured using OMOS.

The Web2ID authentication protocol can also be used by a service provider to prove the ownership of its mashlet. We use this feature as part of authorization delegation protocol that we describe next. The authorization delegation and attribute exchange protocols build upon the authentication protocol described above.

The above basic Web2ID protocol supports user authentication. It can be generalized to support more complex operations such as identity attribute exchange and authorization delegation. In Section 4.5, we will present a mashup relay framework and explain how it facilitates attribute exchange in Web2ID. Our authorization delegation protocol is described in Section 4.6.

Security Analysis of the User Authentication Protocol. Because *Web2ID* uses client-side inter-mashlet communication, its security relies on the client-side communication protocol that is used in its implementation. We assume that the mashlet framework that is used for implementation of *Web2ID* guarantees confidentiality of inter-mashlet communication. This assumption implies that the mashlet framework protects the protocol against MitM attacks by malicious mashlets. Next, we analyze how the user authentication protocol resists against attacks launch by adversaries.

Since the session token χ is encrypted by the public key that is associated with the claimed identity URL (located at the identity page), the user can get access to the session token only if she owns the corresponding private key. Therefore, assuming that only the owner of an identity URL has access to the private key that corresponds to the public key embedded in the corresponding identity page, she will be the only person that can use that session token. This prevents malicious users from forging identities that does not belong to them.

To protect users against relay attacks, *Web2ID* requires service providers to encrypt the domain name of their mashlet besides the session token. This way the identity mashlet can ensure that the mashlet that is requesting the session token is not relaying an encrypted session token issued by another service provider. Finally, since user's credentials and session tokens are never sent over network in clear text, *Web2ID* authentication is immune to passive MitM attacks. As discussed earlier, Web2ID relies on the underlying network protocols (e.g., https) to protect integrity of point to point communication against active MitM attackers. However, to prevent active MitM attackers from replaying a successful authentication transaction, service providers need to record tokens and reject transactions which contain a token which is already used. Number of tokens that need to be saved can be reduced by introducing a timestamp into each token and rejecting tokens that are older than a threshold.

4.5 Attribute Exchange and Relay Mashlet in Web2ID

An important feature supported by most identity management frameworks is that of *attribute exchange*, in which one service provider requests a user's identity attributes (e.g. age) or preferences from another service provider. Attribute exchange protocol allows sensitive applications to bind an identity URL with a physical entity (person) by querying attributes of its owner certified by a trusted third party. For example, SP.com can query certified address of alice.me from a trusted third-party (e.g. Division of Motor Vehicle Bureau or dmv.org) to ensure that URL belongs to the same physical entity that claims its ownership. Attribute exchange is especially important for mashup applications, in which interaction between mashlets is the norm and user may have multiple identities for each service provider as attribute exchange can be used to correlate these identities.

In this section, we introduce a new mashlet-relay structure that enables user-centric client-side communications between two domains. Then, we explain why such a mashlet relay framework is useful in the implementation of identity attribute exchange in Web2ID. The main purpose of mashlet relay is for better user privacy, as it provides an indirect communication channel between two service providers. The providers do not directly interact with each other with regard to a user's request, and thus cannot share information of a user.

4.5.1 Mashlet-Relay Framework

We define *mashlet-relay framework* as a special client-side mashup framework with three mashlets within a browser environment where the communication of two mashlets, each hosting contents of a remote server, is indirect and realized through a third mashlet that is hosted by the local host. We refer to the two mashlets hosted by remote servers as *server mashlets*. A server mashlet also communicates to its corresponding remote server via the mashlet-to-server communication mechanism. We refer to the mashlet that bridges the communication of the two server mashlets as the *relay mashlet*. All inter-mashlet communication follows the mashlet-to-mashlet messaging mechanism. The relay mashlet effectively passes messages between two server mashlets and is able to modify the messages based on user's inputs. Figure 4.3 shows a schematic drawing of such a mashlet relay framework, where the mashlet in the middle (Mediator) mediates the communication between a requester (e.g., SP.com) and a provider (e.g., AttProvider.com). The mediator mashlet is launched by the local host of the individual user. It anonymizes the identity of the requester (e.g., SP.com), as the provider (e.g., AttProvider.com) learns nothing about who issues the request. Such a mashlet relay framework, although simple, supports a user-centric design where the user is able to monitor and actively control the messages being communicated among server mashlets. As a consequence, the client-side relay mashlet eliminates the need of direct communication between the two server mashlets. This feature plays a key role in enabling privacy-aware identity management in Web2ID.

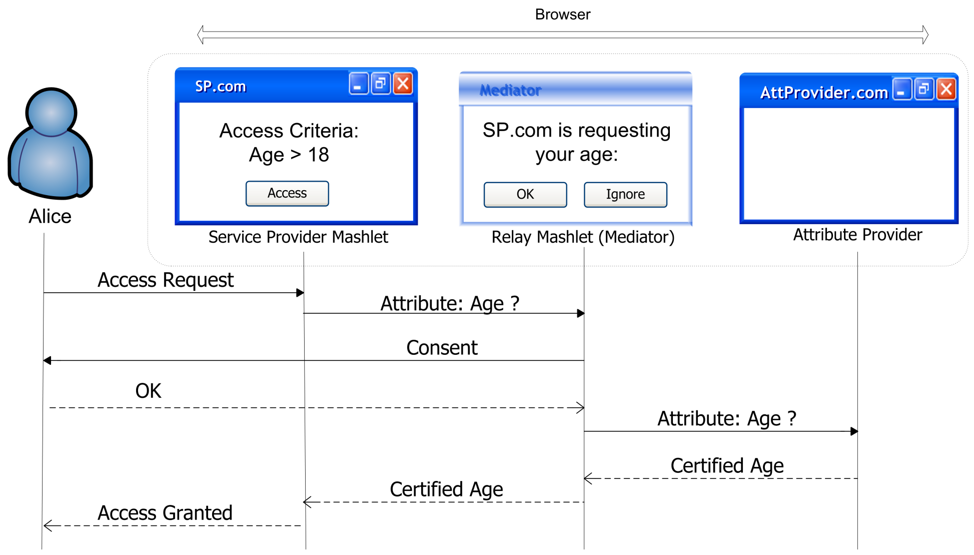

Figure 4.3: Web2ID users mashlet relay communication framework for attribute exchange. In mashlet relay framework, a mashlet (center) mediates the communication between requester (left) and provider (right) and anonymizes the identity of requester.

This mashup-based relay framework naturally facilitates the construction of a privacy-aware identity management protocol, namely identity attribute exchange in SSO, that enables the exchange of user's identity credentials without the direct communication between the identity provider and service provider. In existing (federated) identity management systems, *direct communications* between providers on user's ID information are typically required, which, however, is undesirable as providers may learn sensitive attributes of the user. Therefore, the segregation of providers in their communication protects user privacy and prevents providers from colluding to discover user activities. Yet, in the meantime, proper message exchanges among providers should be allowed, e.g., a service provider may need to verify Alice's identity attributes hosted by an identity provider. Next, we explain why such a mashlet relay framework is useful in the identity attribute exchange in Web2ID.

4.5.2 Identity Attribute Exchange Based on Mashlet Relay

When a service provider requests a user's identity attributes from another service provider, the user may wish to anonymize the identity of the provider requesting these attributes. Doing so prevents the attribute providing service from learning the user's surfing habits. To implement privacy-aware identity attribute exchange, Web2ID avails of the mashup relay framework. In particular, the relay mashlet mediates the exchange of identity attributes between service providers. Because the relay mashlet forwards the request to the attribute provider only after obtaining the user's consent, users have full control over what attributes can be exchanged.

Figure 4.3 presents an example that shows how using *Web2ID* a service provider SP.com can query user's age certified by AttProvider.com. If the attribute requester already knows the user's identity, the identity mashlet of the user can itself be used as a relay mashlet. Alternatively, a mashlet loaded from a trusted third party or the local machine can act as the relay mashlet. We omit the security definition and analysis for our identity attribute exchange protocol, as they can be easily deduced following the analysis in the basic Web2ID protocol.

4.6 Web2ID Extension: Realizing Authorization Delegation

Web application mashlets included in a mashup typically access resources hosted at other domains. In this context, the mashlet that accesses resources is typically called the *Consumer*, while the domain that hosts the resource is called the *Service Provider*. Consumers should not be able to access a user's protected resources unless the user grants them the required access permission.

An *authorization delegation protocol* allows the users to delegate permissions to a consumer to access their resources hosted at a service provider. For example, users may be able to delegate permissions needed to access their files on a photo-sharing website (the service provider) to a website that provides photo editing utilities (the consumer). An authorization delegation protocol should be privacy-preserving in that it must not reveal the user's identity. For instance, users may wish to grant the photo editing service read access to their photos hosted on the photo sharing website without revealing their identity to the photo editing service.

Users may wish to delegate to a consumer the rights to access their resources hosted on a service provider. There are two cases that arise in the implementation of authorization delegation, based upon the privacy guarantees that the user requires.

Case 1: Protecting user identity from the consumer. In the first case, users may not want to disclose their identity to the consumer. For example, a user Alice may wish to print her photos hosted at a photo sharing website SP.com by allowing a printing website Consumer.com to access her photos at SP.com. Yet, she may not wish disclose her identity (i.e. Alice.me) to Consumer.com. To support this case, the authorization delegation protocol should not give any information to the consumer that reveals her identity.

Figure 4.4 illustrates the authorization delegation protocol, via which Consumer.com acquires an opaque token AC to access Alice's resource (e.g., /a/v.jpg) without learning her identity I (e.g., Alice.me). As this figure illustrates, the service provider SP.com uses a secret key SK, known only to the service provider, to generate an opaque token $AC = E_{SK}(\text{Consumer.com, GET, /a/v.jpg}, I)$ that grants Consumer.com read access (i.e., a GET request) to the resource /a/v.jpg, which belongs to I.

When the service provider SP.com receives a request from Consumer.com (via back-end server-to-server communication) containing the access token AC, it first decrypts AC and ensures that that the identity of the requester matches the principal that the token is granted to (Consumer.com); if so, it allows the request.

Case 2: User identity known to consumer In this case, the consumer already knows the user's identity (e.g., because the user has authenticated to the consumer). Figure 4.5 illustrates the protocol used in this case. The identity mashlet of users can independently issue an access delegation certificate using their private keys to grant

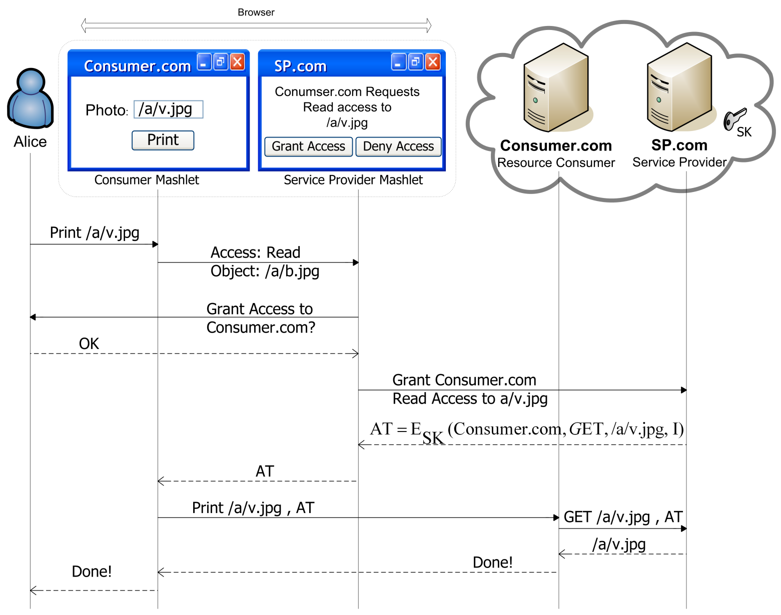
Browser
Consumer.com
Photo: /a/v.jpg
Print
SP.com
Conumser.com Requests Read access to /a/v.jpg
Grant Access
Deny Access
Alice
Consumer Mashlet
Service Provider Mashlet
Consumer.com Resource Consumer
SP.com Service Provider
SK
Print /a/v.jpg
Access: Read Object: /a/b.jpg
Grant Access to Consumer.com?
OK
Grant Consumer.com Read Access to a/v.jpg
AT = E_SK (Consumer.com, GET, /a/v.jpg, I)
AT
Print /a/v.jpg , AT
GET /a/v.jpg , AT
/a/v.jpg
Done!
Done!

Figure 4.4: In Web2ID, a service provider can issue an opaque token to a consumer to access user's resources. In doing so, Web2ID does not reveal the user's identity to the consumer.

the consumer access to their protected resources hosted on a service provider. In turn, the service provider can validate the certificate using the user's public key. The service provider can obtain the public key using the identity URL of the user that the resource belongs to.

Our Web2ID authorization-delegation protocol does not require the consumer to pre-register with the service provider. This property is in sharp contrast to similar protocols (such as OAuth), which require the consumer to pre-register with the service provider. Additionally, Web2ID does not require the service provider or the consumer to maintain protocol-related state during delegation, therefore it is scalable and easy to implement.

Security Analysis. Before serving a request, service providers verify that the access tokens are either issued using their own secret keys or the private key of the owner of the resource. Since these types of tokens can be issued only with user's consent, consumers will not be able to access users resources without agreement of their owner. To prevent MitMs from using h ijacked access tokens, *Web2ID* requires that all access tokens be

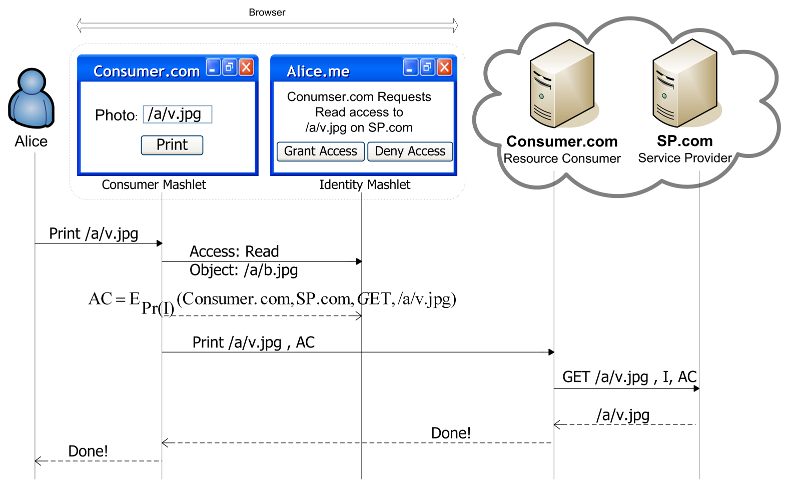

Figure 4.5: The identity mashlet issues a delegation certificate for read access to resource /a/v.jpg. Using this certificate the consumer can access /a/v.jpg on SP.com.

bound to the domain name of the mashlet that the token is granted to. Therefore, these tokens can be used only by the service provider that owns the mashlet. Service providers can use Web2ID authentication to prove ownership of the mashlet that the token is issued for. In the access tokens issued by the service provider, the identity URL of the user is encrypted by service provider's secret key. Therefore, the consumer will not be able to learn the identity of user and this protects the privacy of the user.

4.7 Implementation and Evaluation

Realizing Web2ID requires in-browser symmetric and public-key cryptographic primitives. However, there is no JavaScript cryptographic libraries that provide all the operations that are required for implementation of Web2ID (i.e., HMAC, public-key encryption and public/private key generation). The only JavaScript-based library that implements public-key cryptography [80] does not support public/private key generation, which is required by Web2ID.

As one of the main technical contributions of this paper, we developed a JavaScript-based cryptographic library that not only supports operations that are required by Web2ID but also can be easily extended to support other cryptographic operations.

Our library is fully compatible with commodity browsers, such as IE, Firefox, Chrome, Opera and Safari, and does not require any browser modifications.

4.7.1 Implementation Details

Since development of a JavaScript library from scratch is very time-consuming and error-prone, we based our implementation of the JavaScript cryptographic library on the Java Cryptography Architecture (JCA) [59, 70], an open-source Java-based cryptographic toolkit. We used Google Web Toolkit (GWT) to translate code from Java to JavaScript. However, in implementing this library and porting it to commodity browser platforms, we encountered three technical challenges, namely *performance*, *browser interference*, and *code complexity*, that we describe below.

Performance. Directly compiling the JCA library into JavaScript resulted in extremely poor performance of cryptographic operations. We found that the main performance bottlenecks were BigInteger operations, such as modInverse, mod, and multiplication operations, that are frequently used in cryptographic operations. We addressed this problem by replacing the JCA implementation of BigInteger with the native JavaScript code using the JavaScript Native Interface (JSNI). This replacement significantly improved the performance, with encryption and decryption operations consuming less than a second (see also Section 4.7.2).

Browser Interference. The implementation of the Web2ID protocol requires generation of public/private key pairs when the identity mashlet is first loaded. We observed that key generation algorithms for public-key cryptographic algorithms such as RSA were quite expensive. Because most browsers (and JavaScript interpreters) are single-threaded, users cannot interact with the browser during key generation. Most browsers time out JavaScript functions that execute for long durations of time (typically about 10 seconds). As a result, key generation algorithms are interrupted by the browser.

To overcome browser interference during our key generation operations and keep the browser responsive, we used an *incremental and deferred computation* technique. We observed that the most expensive operation during the generation of public/private

RSA key pairs was the generation of probable prime numbers p and q. The BigInteger.getProbablePrime function continuously generates random odd integers until it finds one that passes Miller-Rabin primality test, thereby resulting in long execution times. We changed this procedure so that each iteration runs in a continuous time slice. We then scheduled the next iteration for another time slice and returned control to the browser, as illustrated in Figure 4.6. This process continues until the key generation algorithm finds a number that passes Miller-Rabin test. We found that this approach was effective at keeping the browser responsive and preventing browser timeouts of JavaScript execution.

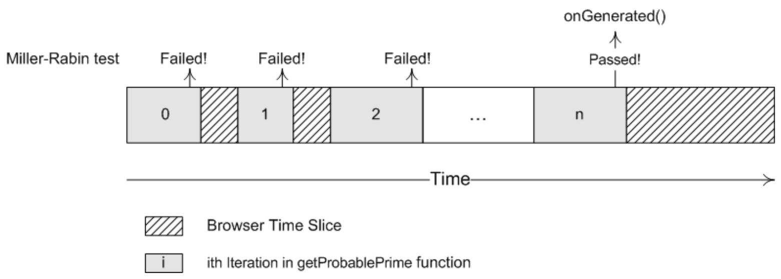

Figure 4.6: Deferred execution of prime number generation.

Code complexity. JCA, upon which our JavaScript library is based, uses several Java features, such as reflection, that are not supported by GWT. Consequently, we first modified JCA to a set of core components that were sufficient to implement cryptographic operations needed for Web2ID. We then used this stripped-down version of JCA with GWT to produce our JavaScript library.

4.7.2 Experimental Results

Our goal is to study feasibility and overhead of using in-browser cryptographic operations. We ran experiments on a machine with the following configuration: Intel Core 2 CPU, 980 MHz, 1.99 GB RAM, Microsoft Windows XP 2002 SP2. We tested our implementation using the following browsers: Google chrome v1.0.154.53 Firefox v3.0.8, Internet Explorer v7.0.5730.13, Opera v9.27, and Apple Safari v3.1.1. The most expensive cryptographic operation that is required by Web2ID is key generation. Figure 4.7 shows the runtime of our RSA keypair generation function for keys of size 512

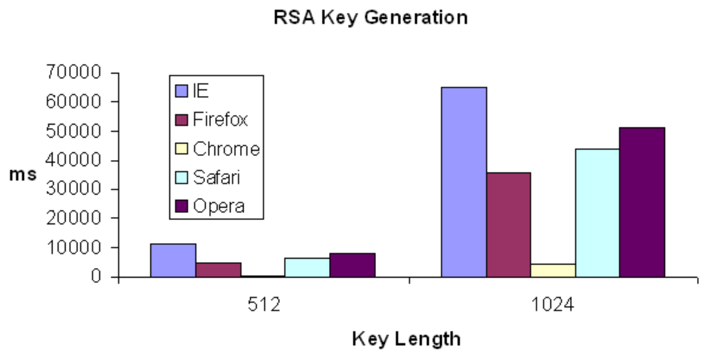

Figure 4.7: Key generation performance in milliseconds of our cryptographic library on different browsers.

and 1024 bits. Since key generation is a probabilistic process, the values reported are averaged results over ten runs. As this Figure shows, Google Chrome, which uses a fast JavaScript Engine (V8), generates a 1024-bit key pair in under 4 seconds. The slowest browser was IE, which took about one minute to generate a 1024-bit key pair. Because key generation is a one-time operation and the browser stays responsive during this time, we feel that this delay is acceptable.

Figure 4.8 shows the performance of RSA encryption/decryption using keys of length 1024 bits. As expected, decryption is more costly compared to encryption and the performance is quite reasonable for web applications. Of the browsers that we tested, Google Chrome had the best performance (less than 100ms for decryption using 1024-bit key).

4.8 Applying *Web2ID* to Web-based Desktop Applications

In this section, we present an application of the *Web2ID* protocol to a web-based desktop application (in short, a *Webtop*). Web-based desktop applications (or webtops) provide a desktop-like environment within the browser. Users can open multiple office applications within a webtop, and can easily share data between these applications (e.g., using drag-and-drop). A number of popular Webtops are now available, including Glide OS, eyeOS, and G.ho.st [49, 43, 48].

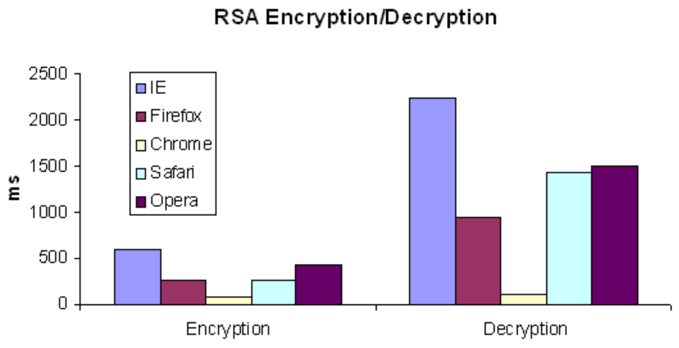

Figure 4.8: The performance in milliseconds of RSA encryption and decryption on different browsers.

To demonstrate the application of *Web2ID* to Webtops, we build a Webtop application called *Zaranux* [91]. To the best of our knowledge, this application is the first linux-based open-source Webtop. Zaranux emulates a desktop environment, such as Gnome or KDE, within the Web browser. It provides several applications, including a command-line interface (i.e., a terminal), via which users can easily browse and access their remote file system, upload/download files, and run third-party applications. Each third-party application, such as a word processor, runs within its own protection domain and can access user data in a controlled manner after obtain the user's consent.

Because Webtops support a variety of office applications, typically from different sources, users often have to authenticate themselves with each such application. A Webtop that integrates an identity management solution can therefore greatly improve end-user experience. However, existing identity management solutions are not directly applicable to Webtop environments due to the heavy use of redirection and privacy concerns. Office applications are typically stateful and contain unsaved data. The identity management solutions implemented via a series of HTTP redirections would result in the loss of unsaved data.

We therefore integrated our implementation of *Web2ID* with Zaranux. Below, we discuss how *Web2ID* provides single sign-on, authorization delegation and resource access in Zaranux.

User Authentication and Single Sign-On. A user first logs into Zaranux and enters his credentials into a mashlet provided by an identity provider (as discussed in Section 4.4). The implementation of *Web2ID* in Zaranux ensures that any applications that require authentication can seamlessly verify the identity of the user without requiring the user to authenticate again. Zaranux shares the identity of its users with applications only after getting their consent. The example below explains a common authentication scenario.

Suppose that Alice has logged into Zaranux, and has started a financial application, e.g., located at the URL http://investment.com. When Alice tries to access her data at investment.com, it must authenticate her. To do so, the mashlet from investment.com requests Alice's identity URL by making a client-side system call to Zaranux. After getting Alice's consent, Zaranux returns her identity to investment.com. In turn, according to *Web2ID* protocol, to verify this claim, investment.com mashlet forwards the claimed URL to the investment.com server, which retrieves the public key from Alice's identity URL, encrypts a session token and returns it to the client side (as in the *Web2ID* protocol). Note that all these steps are transparent to Alice, once she has authenticated herself to Zaranux, which in turn provides identity management services to other office applications that require authentication.

Authorization Delegation and Resource Access. In Zaranux, an office application that wishes to access a resource, such as a file or directory, invokes a client-side API akin to the open system call on traditional desktop operating systems. This API call returns a file handle that can be used to access the resource. This file handle serves as an opaque capability token that delegates a certain access permission (e.g., read, write or delete) to the token holder. Zaranux also implements the authorization delegation protocol (discussed in Section 4.6), and uses relays to enable delegation in a privacy-preserving manner.

4.9 Related Work

OpenID implements decentralized user authentication on the Internet via a series of HTTP redirections within the user's browser. These redirections perform inter-domain communication between the IdP and SP and transmit the user's credentials from the IdP to the SP. However, redirections are ill-suited for stateful Ajax-based applications, such as Web desktops and Web-based office applications, because they involve unloading/reloading the application upon each redirection. Without application-level support, unloading/reloading operations will result in the loss of unsaved data. In addition, the use of an identity provider to manage credentials and personal information raises privacy concerns. Web2ID provides technical solutions for both problems.

Our Web2ID protocol can be realized with any secure mashup frameworks. They provide general infrastructure and environments for content providers to communicate in our identity management applications. There have been a couple of recent work that proposed secure mashup solutions including MashupOS [83], SMash [63], PostMessage method [22], and OMOS [90]. The main goal of these solutions is two-fold: to isolate contents from different sources in sandbox structures such as frames and to achieve frame-frame communication.

SMash [63] uses the concepts in publish-subscribe systems and creates an efficient event hub abstraction that allows the mashup integrator to securely coordinate and manage contents and information sharing from multiple domains. SMash mashup integrator (i.e., the event hub) is assumed to be trusted by all the web services. MashupOS [83] applies concepts in operating systems in mashup and develops sophisticated browser extensions and environments that enable the separation and communication of frames similar to inter-process communication management in the operating system. As mentioned earlier, the OpenMashupOS (OMOS) framework contains a key-based protocol providing secure frame-to-frame communication [90].

Despite the recent progress on mashup applications, the identity management in mashup environments has not been systematically investigated in the literature. Camenisch *et al.* presented the architecture of PRIME (Privacy and Identity Management for Europe), which implements a technical framework for processing personal data [33]. PRIME focuses on enabling users to actively manage and control the release of their private information. Privacy policies for liberty single sign-on [34, 66] have been presented [73] by Pfitzmann. The paper identifies a number of privacy ambiguities in Liberty V1.0 specifications [67] and propose privacy policies for resolving them. A good article on the issues and guidelines for user privacy in identity management systems was written by Hansen, Schwartz, and Cooper [55].

In the federated identity management (FIM) solution by Bhargav-Spantzel *et al.*, personal data such as a social security number is never transmitted in cleartext to help prevent identity theft [25]. Commitment schemes and zero-knowledge proofs are used to commit data and prove the knowledge of the data. BBAE is the federated identity-management protocol proposed by Pfitzmann and Waidner [75]. They gave a concrete browser-based single sign-on protocol that aims at the security of communications and the privacy of user's attributes. Goodrich *et al.* proposed a notarized FIM protocol that uses a trusted third-party, called notary server, to effectively eliminate the direct communication between identity provider and service provider [50]. The main difference with these proposed privacy-aware ID management solutions and our approach is that we study ID management in the client-side mashup environment through a novel and efficient mashlet relay framework.

In the access control area, the closest work to ours is the framework for regulating service access and release of private information in web-services by Bonatti and Samarati [29]. They study the information disclosure using a language and policy approach. We designed cryptographic solutions to control and manage information exchange. Another related work aiming to protect user privacy in web-services is the point-based trust management model [87], which is a quantitative authorization model. Point-based authorization allows a consumer to optimize privacy loss by choosing a subset of attributes to disclose based on personal privacy preferences. The above two models mainly focus on the client-server model, whereas our architecture include two different types of providers.

4.10 Summary

As mashup applications increase in popularity, we expect that they will also be used with sensitive Web services, such as financial and banking applications. When mashups are used in such scenarios, it is important to provide features such as identity management. We presented Web2ID, an identity management protocol for mashup applications. Web2ID preserves the privacy of the end user and eliminates the need for a trusted identity provider in the online single sign-on process. We described how this feature can be realized with conventional public-key cryptography. We also described a mashlet-relay framework that enables efficient yet indirect communication between two server mashlets via a local relay mashlet controlled by the user. Such a relay framework allows for attribute exchange without disclosing the user's surfing habits to service providers. Our implementation of Web2ID and the relay framework is implemented as an in-browser library and is fully compatible with commodity browsers.

We overcome several technical difficulties and successfully implemented a public-key cryptographic JavaScript library for the browser to perform cryptographic operations such as key-pair generation, encryption, and decryption. This technical contribution is beyond the specific identity management problem studied. Last but not the least, we also described how Web2ID applies to the emergent Webtop environments.

Chapter 5

Conclusions and Future Work

Cloud computing is an emerging paradigm which its cost-effectiveness and flexibility have given it a tremendous momentum. However, there are many security challenges that, if not addressed well, may impede its fast adoption and growth. This dissertation primarily addresses the problem of sharing, managing and controlling access to sensitive resources and services in an integrated cloud environment.

The primary conclusion of our research is that adoption of user-centric security models and shifting certain parts of communication and computation to the client side allows us to provide the cloud consumers with more visibility and control over their resources. Therefore, using this approach not only the security and privacy concerns of cloud consumers can be addressed more effectively, but also the burden of managing end-users' identities and fine-granular access control will be reduced from cloud service providers.

In this dissertation, we presented our client-side access control protocol called *K2C*. This protocol illustrates how recent cryptographic schemes can be utilized to develop an effective client-side access control protocol for protecting confidentiality and integrity of data stored in an untrusted cloud storage. Our protocol protects the privacy of end-users by hiding their identities, as the authentication is realized based on attributes or properties of a user, as opposed to the identity.

We also introduced our client-side integration framework called *OMOS*. This framework abstracts away all the details of inter-domain communication and provides the developers with a powerful layered communication stack for component-based client-side integration. On top of *OMOS* framework, we introduced a new identity management protocol called *Web2ID*, which is specifically tailored for mashup applications with

AJAX-based architecture. We showed how using client-side communication and asymmetric cryptography, we can eliminate the need for identity providers, and thus protect the privacy of end-users during authentication.

Below we present the list of related open problems and opportunities for extending and improving the frameworks and protocols that we presented in this dissertation:

- **Off-loading *K2C* key management to cloud.** In current design of *K2C* protocol clients are responsible for key management. With respect to the capabilities of clients, *K2C* is based on two assumptions: The clients have access to 1) local storage with a mechanism for securely storing/retrieving the keys and its metadata. 2) point-to-point secure communication channels for distributing secret keys. Relaxing these assumptions can improve usability and security of *K2C*. The key challenge here is to do so without increasing the trust requirement on storage provider. Development of techniques such as proxy re-encryption [19] is very promising for solving these types of problems in clouds [88].

- **Using cloud for client-side cryptography.** Both *K2C* and *Web2ID* require some expensive client-side cryptography. The required computational power may limit their usage in certain domains (e.g. lightweight wireless devices). Finding a secure way to leverage cloud computational power for performing the required mathematical computations can alleviate this problem. Zhou *et al.* in [93] introduce and analyze some techniques that one may be able to adopt to address these limitations.

- **Anonymous client-side communication.** As part of *Web2ID*, we introduced a *relay mashup framework*, in which a trusted client-side *relay agent* mediates the communication between two mashlets to anonymizes the service provider and/or the service consumer. Using this technique, without requiring any complex cryptography, we were able to design a privacy-preserving identity attribute exchange protocol. However, in this solution the relay mashlet needs to be hosted on the client host, a requirement that limits the portability of our solution. Designing a new primitive or enhancing `postMessage` to support anonymous client-side communication will make these kinds of solutions more portable and usable.

- **Backend authentication in presence of untrusted client.** In *Web2ID* our focus was mainly on authentication of the end users. Another related challenging problem is the problem of authentication between backend service providers in presence of an untrusted client. To our knowledge currently there is no protocol that addresses this problem without relying on trusted third-parties. Recently a mulit-party protocol called MashSSL [14] has been proposed to address this problem but it relies on a trusted third party to issue certificates for backend servers.

The above items provide some directions for extending our frameworks and improving their usabilities.

Vita

Saman Zarandioon

EDUCATION

2006-12	Ph.D. in Computer Science, Rutgers University
2004-05	M.Sc. in Computer Science, NJIT
1995-01	B.Sc. in Computer Science and Engineering, Bahaí Institute for Higher Education (BIHE)

EMPLOYMENT

2011-2012	Software Engineer, Elastic Cloud Computing, Amazon Inc.
2007-2011	Senior Integration Engineer, Middleware Engineering, MetLife Inc.
2005-2007	Technical Analyst, Business Integration, Colgate-Palmolive Inc.

PEER-REVIEWED PAPERS

- Saman Zarandioon, Danfeng Yao, Vinod Ganapathy, “K2C: Cryptographic Cloud Storage With Lazy Revocation and Anonymous Access.”, *In SecureComm’11: Proceedings of the* 7^{th} *International ICST Conference on Security and Privacy in Communication Networks, London, UK, September 2011, Volume 96 of LNICST, Springer, pages 491-510. Acceptance rate 24.2%.*
- Saman Zarandioon, Danfeng Yao, Vinod Ganapathy, “Privacy-aware identity management for client-side mashup applications”, *In DIM’09: Proceedings of the* 5^{th} *ACM Workshop on Digital Identity Management, Chicago, Illinois, November 2009, ACM Press, pages 21-30.*
- Saman Zarandioon Danfeng Yao, Vinod Ganapathy, “OMOS: A Framework for Secure Communication in Mashup Applications”, *In ACSAC’08: Proceedings of the* 24*th Annual Computer Security Applications Conference, Anaheim, California, USA, December 2008, IEEE Computer Society Press, pages 355-364. Acceptance rate 24.8%.*
- Saman Zarandioon, Alexander Thomasian, “Optimization of Online Disk Scheduling Algorithms”, *ACM SIGMETRICS Performance Evaluation Review - Storage Systems, Volume 33 , Issue 4 (March 2006), pages: 42-46.*

References

[1] RFC 3174, US Secure Hash Algorithm 1 (SHA1) http://www.ietf.org/rfc/rfc3174.txt.

[2] OpenID Specification. http://openid.net/developers/specs/.

[3] Philip Zimmermann, http://www.philzimmermann.com.

[4] RFC 4252, The Secure Shell (SSH) Authentication Protocol http://tools.ietf.org/html/rfc4252.

[5] 104th United States Congress. Health Insurance Portability and Accountability Act of 1996 (HIPPA), http://aspe.hhs.gov/admnsimp/pl104191.htm.

[6] Amazon S3. http://aws.amazon.com/s3/.

[7] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[8] Google App Engine. http://appengine.google.com.

[9] Google Juice. http://code.google.com/p/google-guice/.

[10] HIBE Crypto Library. https://sourceforge.net/projects/hibe.

[11] jPBC. http://gas.dia.unisa.it/projects/jpbc.

[12] KP-ABE Crypto Library. https://sourceforge.net/projects/kpabe.

[13] Lisp S-expression parser. http://sourceforge.net/projects/jsexp.

[14] MashSSL. http://mashssl.org.

[15] Open Source Implementation of CP-ABE. http://acsc.cs.utexas.edu/cpabe/.

[16] SQL Data Services/Azure Services Platform. http://http://www.windowsazure.com.

[17] Dave Ahmad. The confused deputy and the domain hijacker. *IEEE Security and Privacy*, 6(1):74–77, January/February 2008.

[18] Asynchronous JavaScript and XML Tutorials. http://developer.mozilla.org/ en/-docs/AJAX.

[19] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *NDSS*, pages 29–43, 2005.

[20] Michael Backes, Christian Cachin, and Alina Oprea. Secure Key-Updating for Lazy Revocation. In *Research Report RZ 3627, IBM Research*, pages 327–346. Springer, 2005.

[21] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[22] Adam Barth, Collin Jackson, and John C. Mitchell. Securing Browser Frame Communication. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[23] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, CCS '93, pages 62–73. ACM, 1993.

[24] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.

[25] Abhilasha Bhargav-Spantzel, Anna Cinzia Squicciarini, and Elisa Bertino. Establishing and Protecting Digital Identity in Federation Systems. *Journal of Computer Security*, 14(3):269–300, 2006.

[26] Marina Blanton. Key Management in Hierarchical Access Control Systems, 2007. PhD Thesis, Purdue University, Aug. 2007.

[27] Marina Blanton, Nelly Fazio, and Keith B. Frikken. Dynamic and Efficient Key Management for Access Hierarchies. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2005.

[28] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT*, pages 127–144. Springer-Verlag, 1998.

[29] Piero A. Bonatti and Pierangela Samarati. A Uniform Framework for Regulating Service Access and Information Release on the Web. *Journal of Computer Security*, 10(3):241–272, 2002.

[30] Dan Boneh and Matthew Franklin. Identity-based encryption from the weil pairing. *SIAM J. Comput.*, 32:586–615, March 2003.

[31] Jose M. Alcaraz Calero, Nigel Edwards, Johannes Kirschnick, Lawrence Wilcock, and Mike Wray. Toward a multi-tenancy authorization system for cloud services. *IEEE Security and Privacy*, 8:48–55, 2010.

[32] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In *ACM Computer and Communication Security 2002*. ACM, 2002.

[33] Jan Camenisch, Abhi Shelat, Dieter Sommer, Simone Fischer-Hübner, Marit Hansen, Henry Krasemann, G. Lacoste, Ronald Leenes, and Jimmy Tseng. Privacy and Identity Management for Everyone. In *Proceedings of the 2005 ACM Workshop on Digital Identity Management*, pages 20–27, November 2005.

[34] S. Cantor, F. Hirsch, J. Kemp, R. Philpott, E. Maler, J. Hughes, J. Hodges, P. Mishra, and J. Moreh. Security Assertion Markup Language (SAML) V2.0. Version 2.0. OASIS Standards.

[35] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th symposium on Operating systems design and implementation - volume 7*, pages 205–218, 2006.

[36] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 85–90, New York, NY, USA, 2009. ACM.

[37] CrossFrame, JavaScript Yahoo API.

[38] Steven Crites, Francis Hsu, and Hao Chen. Omash: Enabling secure web mashups via object abstractions. In *15 ACM Conference on Computer and Communicatios Security*, 2008.

[39] Joan Daemen and Vincent Rijmen. Rijndael/aes. In *Encyclopedia of Cryptography and Security*. 2005.

[40] DOJO Library. Part 5. http://dojotoolkit.org/book/dojo-book-0-4/.

[41] Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). In *RFC3147*.

[42] Robert Ennals and Minos Garofalakis. MashMaker: mashups for the masses. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1116 – 1118. ACM, 2007.

[43] eyeOS, http://eyeos.org/.

[44] Facebook API. http://developers.facebook.com/.

[45] Kevin Fu. Group sharing and random access in cryptographic storage file systems. Technical report, Masters thesis, MIT, 1999.

[46] Kevin Fu, Seny Kamara, and Tadayoshi Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *NDSS*, 2006.

[47] Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In *ASIACRYPT*, pages 548–566, 2002.

[48] G.ho.st, http://g.ho.st/.

[49] Glide OS, http://www.glidedigital.com.

[50] Michael T. Goodrich, Roberto Tamassia, and Danfeng (Daphne) Yao. Notarized federated ID management and authentication. *Journal of Computer Security*, 16(4):399–418, 2008.

[51] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.

[52] Chris Grier, Shuo Tang, and Samuel T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, May 2008.

[53] Dominik Grolimund, Luzius Meisser, Stefan Schmid, and Roger Wattenhofer. Cryptree: A folder tree structure for cryptographic file systems. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.

[54] Steve Hanna, E C R Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. *The Emperors New APIs: On the (In) Secure Usage of New Client-side Primitives.* 2010.

[55] Marit Hansen, Ari Schwartz, and Alissa Cooper. Privacy and Identity Management. *IEEE Security and Privacy*, 6(2):38–45, 2008.

[56] Hyunuk Hwang, Gyeok Jung, Kiwook Sohn, and Sangseo Park. A study on mitm (man in the middle) vulnerability in wireless network using 802.1x and eap. In *ICISS '08: Proceedings of the 2008 International Conference on Information Science and Security*, pages 164–170, Washington, DC, USA, 2008. IEEE Computer Society.

[57] Collin Jackson and Adam Barth. Beware of finer-grained origins. In *W2SP 2008: Web 2.0 Security and Privacy. Held in conjunction with the 2008 IEEE Symposium on Security and Privacy*, 2008.

[58] Collin Jackson and Helen J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *Proceedings of the 16th International Conference on World Wide Web*, pages 611–620, 2007.

[59] Java Cryptography Architecture. http://java.sun.com/j2se/1.4.2/docs/guide/ security/CryptoSpec.html.

[60] Eu jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. Sirius: Securing remote untrusted storage. In *NDSS*, pages 131–145, 2003.

[61] JSON-RPC 1.1 Specification. http://json-rpc.org/wd/JSON-RPC-1-1-WD-20060807.html.

[62] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage, 2003.

[63] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure Component Model for Cross-Domain Mashups on Unmodified Browsers. In *Proceedings of the 17th International Conference on World Wide Web*, 2008.

[64] Krawczyk, Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. In *RFC2104*.

[65] Kapil Singh Wenke Lee. On the design of a web browser: Lessons learned from operating systems. In *W2SP 2008: Web 2.0 Security and Privacy. Held in conjunction with the 2008 IEEE Symposium on Security and Privacy*, 2008.

[66] Liberty Alliance Project. http://www.projectliberty.org.

[67] July 2002. Liberty Alliance Project: Liberty Protocols and Schemas Specification, Version 1.0.

[68] Microsoft Windows Live Contacts. htp://dev.live.com/mashups/trypresencecontrol/.

[69] NIST. http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf.

[70] OpenJDK, http://openjdk.java.net/.

[71] OpenMashupOS, http://omos.zaranux.com/.

[72] OpenSocial API. http://code.google.com/apis/opensocial/.

[73] Birgit Pfitzmann. Privacy in Enterprise Identity Federation - Policies for Liberty Single Signon. In *Proceedings of the Third International Workshop on Privacy Enhancing Technologies (PET 2003)*, volume 2760, pages 189–204, 2003.

[74] Birgit Pfitzmann and Michael Waidner. Privacy in browser-based attribute exchange. In *Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pages 52–62. ACM, 2002.

[75] Birgit Pfitzmann and Michael Waidner. Federated Identity-Management Protocols. In *Security Protocols Workshop*, pages 153–174, 2003.

[76] R. Leenes, J. Schallabck, and M. Hansen. Privacy and identity management for europe. Prime whitepaper, 15 May 2008.

[77] Andrzej M. Goscinski Rajkumar Buyya, James Broberg. *Cloud Computing: Principles and Paradigms*. Wiley, 2011.

[78] Erik Riedel, Mahesh Kallahalla, and Ram Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[79] Same Origin Policy. http://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

[80] RSA JS library. http://www-cs-students.stanford.edu/ tjw/jsbn/.

[81] Paul Stanton, William Yurcik, and Larry Brumbaugh. Protecting multimedia data in storage: A survey of techniques emphasizing encryption. In *IS and T/SPIE International Symposium Electronic Imaging / Storage and Retrieval Methods and Applications for Multimedia*, pages 18–29, 2005.

[82] Hassan Takabi, James B.D. Joshi, and Gail-Joon Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security and Privacy*, 8:24–31, 2010.

[83] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *ACM Symposium on Operating Systems Principle (SOSP)*, pages 1–16. ACM Press, 2007.

[84] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 355–370, Berlin, Heidelberg, 2009. Springer-Verlag.

[85] A. Whitten and J.D. Tygar. Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In *8th Usenix security symposium*, pages 169–184, 1999.

[86] Huijun Xiong, Xinwen Zhang, Wei Zhu, and Danfeng Yao. CloudSeal:End-to-End Content Protection in Cloud-based storage and delivery services. In *Securecomm*, 2011.

[87] Danfeng Yao, Keith B. Frikken, Mikhail J. Atallah, and Roberto Tamassia. Point-Based Trust: Define How Much Privacy Is Worth. In *Proc. Int. Conf. on Information and Communications Security (ICICS)*, volume 4307 of *LNCS*, pages 190–209. Springer, 2006.

[88] Shucheng Yu, Cong Wang, Kui Ren, and Wenjing Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 534–542, Piscataway, NJ, USA, 2010. IEEE Press.

[89] Saman Zarandioon, Danfeng Yao, and Vinod Ganapathy. K2C: Cryptographic Cloud Storage With Lazy Revocation and Anonymous Access. Technical report, Rutgers University. DCS-tr-688.

[90] Saman Zarandioon, Danfeng Yao, and Vinod Ganapathy. OMOS: A Framework for Secure Communication in Mashup Applications. In *ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference*, December 2008.

[91] Zaranux, http://zaranux.com/, Saman Zarandioon.

[92] Zaranux Open Source Project, http://zaranux.sourceforge.net/.

[93] Zhibin Zhou and Dijiang Huang. Efficient and secure data storage operations for mobile cloud computing. Cryptology ePrint Archive, Report 2011/185, 2011. http://eprint.iacr.org/.