# Secure Data Preservers for Web Services

Jayanthkumar Kannan, *Google Inc.*
Petros Maniatis, *Intel Labs*
Byung-Gon Chun, *Yahoo! Research*

## Abstract

We examine a novel proposal wherein a user who hands off her data to a web service has complete choice over the code and policies that constrain access to her data. Such an approach is possible if the web service does not require raw access to the user's data to implement its functionality; access to a carefully chosen interface to the data suffices.

Our data preserver framework rearchitects such web services around the notion of a *preserver*, an object that encapsulates the user's data with code and policies chosen by the user. Our framework relies on a variety of deployment mechanisms, such as administrative isolation, software-based isolation (e.g., virtual machines), and hardware-based isolation (e.g., trusted platform modules) to enforce that the service interacts with the preserver only via the chosen interface. Our prototype implementation illustrates three such web services, and we evaluate the cost of privacy in our framework by characterizing the performance overhead compared to the status quo.

## 1 Introduction

Internet users today typically entrust web services with diverse data, ranging in complexity from credit card numbers, email addresses, and authentication credentials, to datasets as complex as stock trading strategies, web queries, and movie ratings. They do so with certain expectations of *how* their data will be used, *what* parts will be shared, and with *whom* they will be shared. These expectations are often violated in practice; the Dataloss database [1] lists 400 data loss incidents in 2009, each of which exposed on average half a million customer records outside the web service hosting those records.

Data exposure incidents can be broadly categorized into two classes: *external* and *internal*. External violations occur when an Internet attacker exploits service vulnerabilities to steal user data. Internal violations occur when a malicious insider at the service abuses the possession of user data beyond what the user signed up for, e.g., by selling customer marketing data. 65% of the aforementioned data-exposure incidents are external, while about 30% are internal (the remainder have no stated cause).

The impact of such data exposure incidents is exacerbated by the fact that data owners are powerless to *proactively* defend against the possibility of abuse. Once a user hands off her data to a web service, the user has *given up control* over her data irreversibly; "the horse has left the barn" forever and the user has no further say on how her data is used and who uses it.

This work introduces a novel proposal that restores control over the user's data to the user; we insist that *any code* that needs to be trusted by the user and *all* policies on how her data is accessed are *specified* by her. She *need not* rely on the web service or any of its proprietary code for performing access control over her data; the fate of her data is completely up to her. The user is free to choose code from any third party (e.g., an open source repository, or a security company) to serve as her software trusted computing base; she need not rely on a proprietary module. This trust can even be based on a proof of correctness (e.g., Proof Carrying Code [20]). Further, *any policies* pertaining to how her data is stored and accessed (e.g., by whom, how many times) are under her control. These properties of personalizable code and policies distinguish us from currently available solutions.

At first look, personalizable trust looks difficult to achieve: the web service may have arbitrary code that requires access to the user's data, but such code cannot be revealed publicly and the user cannot be allowed to choose it. However, for certain classes of web services where the application does not require access to the raw data and a *simple interface* suffices, such personalizable trust is possible. We will show later that for several classes of web services such restricted access is sufficient.

For such services, it is possible to enforce the use of a *well-defined, access-controlled interface* to user data. For example, a movie recommendation service that requires only statistical aggregates of a user's movie watching history need never access the user's detailed watching history. To provide such enforcement, we leverage the idea of preserving the user's own data with code and policy she trusts; we refer to such a preserved object as a *secure data preserver* (see Figure 1).

One could imagine a straightforward implementation of the secure data preserver (SDaP) approach that serves the user's data via the interface from a trusted server (say the user's home machine, or a trusted hosting service). Unfortunately, this solution has two main drawbacks. First, placing user data at a maximally isolated hosting service may increase provisioning cost and access latency, and reduce bandwidth, as compared to the status quo which offers no isolation guarantees. Second, even when data placement is explicitly decided by a particular business

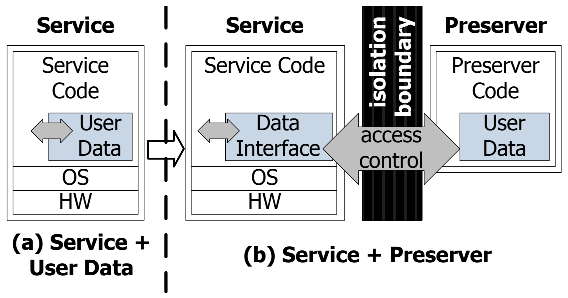

**Figure 1:** The SDaP approach.

model—for instance, charging services such as Google Checkout that store user data at Google's data centers—the user control available may be lacking: Google Checkout only offers a simple, click-to-pay interface, leaving the user with limited access-control options.

In this work, we explore a general software architecture to enable the use of preservers by users and web services. Our architecture has three salient features. First, it supports *flexible* placement and hosting decisions for preservers to provide various trade-offs between performance, cost, and isolation. Apart from the options of hosting the preserver at a client's machine or at a trusted third party, our framework introduces the *colocation* option; the preserver can be hosted securely at the service itself by leveraging a trusted hardware or software module (e.g., in a separate virtual machine). This colocation option presents a different trade-off choice; higher performance, zero provisioning cost on part of the user, and the involvement of no third parties. Second, we provide a *policy layer* around the available data placement mechanisms with two purposes: to mediate *interface access* so as to enable fine-grained control even for coarse-grained interfaces, and to mediate *data placement* so as to fit the user's desired cost, isolation, and performance goals. Third, we offer preserver *transformation mechanisms* that reduce the risk of data exposure (by filtering raw data) or increase anonymity (by aggregating data with those of other users).

We evaluate SDaPs experimentally in a prototype that implements three applications with diverse interface access patterns: day-trading, targeted advertising, and payment. We evaluate these applications along a spectrum of placement configurations: at the user's own computer or at a trusted third party (TTP), offering maximum isolation from the service but worst-case performance, and at the service operator's site, isolated via virtualization, offering higher performance. The isolation offered by client- or TTP-hosted preservers comes at the cost of Internet-like latencies and bandwidth for preserver access (as high as 1 Gbps per user for day-trading). Our current virtual machine monitor/trusted platform module based colocation implementation protects against external threats; it can be extended to resist internal threats as well by leveraging secure co-processors using standard techniques [32]. Colocation offers significantly better performance than off-site placement, around 1 ms interface latency at a reasonable storage overhead (about 120 KB per user). Given the growing cost of data exposure (estimated to be over 200 dollars per customer record [22]), these overheads may be an acceptable price for controls over data exposure, especially for financial and health information.

While we acknowledge that the concept of encapsulation is itself well-known, our main contributions in this work are: (a) introducing the principle of interface-based access control in the context of web services, and demonstrating the wide applicability of such a model, (b) introducing a new colocation model to enforce such access control, (c) design of a simple yet expressive policy language for policy control, and (d) an implementation that demonstrates three application scenarios.

Our applicability to a service is limited by three factors. First, SDaPs make sense only for applications with simple, narrow interfaces that expose little of the user data. For rich interfaces (when it may be too optimistic to hope for a secure preserver implementation) or interfaces that may expose the user's data directly, information flow control mechanisms may be preferable [12, 24, 27, 33]. Second, we do not aim to *find* appropriate interfaces for applications; we limit ourselves to the easier task of helping a large class of applications that already have such interfaces. Third, we recognize that a service will have to be refactored to accommodate strong interfaces to user data, possibly executing locally preserver code written elsewhere. We believe this is realistic given the increasing support of web services for third-party applications.

The rest of the paper is structured as follows. Section 2 refines our problem statement, while Sections 3 and 4 present an architectural and design overview respectively of our preserver framework. We discuss our implementation in Section 5 and evaluate its performance and security in Section 6. We then present related work in Section 7 and conclude in Section 8.

## 2 The User-Data Encapsulation Problem

In this section, we present a problem statement, including assumptions, threat models we target, and the deployment scenarios we support. We then present motivating application scenarios followed by our design goals.

### 2.1 Problem Statement

Our goal is to rearchitect web services so as to isolate user data from service code and data. Our top priority is to protect user data from *abusive access*; we define as abusive any access that violates a well-defined, well-known interface, including unauthorized disclosure or update. We aim for the interface to be such that the data need never be revealed directly and policies can be meaningfully specified on its invocation; we will show such interfaces can be found for a variety of services. A secondary priority is to provide reasonable performance and scalability. Given a chosen interface, our security objective is to ensure that any information revealed to a service is obtainable only via a sequence of legal interface invocations. This significantly limits an adversary's impact since the interface does not expose raw data; at best, she can exercise the interface to the maximum allowed by the policies.

It is important to clarify the scope of our work. First, we assume that an interface has been arrived at for a service that provides the desired privacy to the user; we do not verify that such an interface guarantees the desired privacy, or automate the process of refactoring existing code. Second, we assume that a preserver implementation carries out this interface correctly without any side channels or bugs. We aim for simple narrow interfaces to make correct implementation feasible.

## 2.2 Threat Model

The threat model we consider is variable, and is chosen by the user. We distinguish three choices, of increasing threat strength. A *benign threat* is one caused by service misconfiguration or buggy implementation (e.g., missing access policy on an externally visible database, insecure cookies, etc.), in the absence of any malice. An *external adversarial threat* corresponds to an Internet attacker who exploits software vulnerabilities at an otherwise honest service. An *internal adversarial threat* corresponds to a malicious insider with physical access to service infrastructure.

We make standard security assumptions. First, trusted hardware such as Trusted Platform Modules (TPM [5]) and Tamper Resistant Secure Coprocessors (e.g., IBM 4758 [11]) provide their stated guarantees (software attestation, internal storage, data sealing) despite software attacks; tamper-resistant secure coprocessors can also withstand physical attacks (e.g., probes). Second, trusted hypervisors or virtual machine monitors (e.g., Terra [13], SecVisor [25]) correctly provide software isolation among virtual machines. Third, a trusted hosting service can be guaranteed to adhere to its stated interface, and is resistant to internal or external attacks. Finally, we assume that standard cryptography works as specified (e.g., adversaries cannot obtain keys from ciphertext).

## 2.3 Deployment Scenario

We aim to support three main deployment scenarios corresponding to different choices on the performance-isolation trade-off.

The closest choice in performance to the status quo with an increase in isolation keeps user data on the same service infrastructure, but enforces the use of an access interface; we refer to this as *colocation*. Typical software encapsulation is the simplest version of this, and protects from benign threats such as software bugs. Virtualization via a secure hypervisor can effectively enforce that isolation even for external attacks that manage to execute at the privilege level of the service. Adding attestation (as can be provided by TPMs for instance) allows an honest service to prove to clients that this level of isolation is maintained. However, internal attacks can only be tolerated with the help of tamper-resistant secure co-processors. Though our design focuses primarily on the colocation scenario, we support two other deployment options.

The second deployment option is the *trusted third party (TTP)* option; placing user data in a completely separate administrative domain provides the greatest isolation, since even organizational malfeasance on the part of the service cannot violate user-data interfaces. However, administrative separation implies even further reduced interface bandwidth and timeliness (Internet-wide connections are required). This scenario is the most expensive for the user, who may have to settle for existing types of data hosting already provided (e.g., use existing charging services via their existing interfaces), rather than springing for a fully customized solution. The third deployment option offers a similar level of isolation and is the cheapest (for the user); the user can host her data on her own machine (the *client-side* option). Performance and availability are the lowest due to end-user connectivity.

## 2.4 Usage Idioms

The requirement of a suitable interface is fundamental to our architecture; we now present three basic application *idioms* for which such interface-based access is feasible to delineate the scope of our work.

*Sensitive Query Idiom:* Applications in this idiom are characterized by a large data stream offered by the service (possibly for a fee), on which the user wishes to evaluate a sensitive query. Query results are either sent back to the user and expected to be limited in volume, or may result in service operations. For example, in Google Health, the user's data is a detailed list of her prescriptions and diseases, and the service notifies her of any information relating to these (e.g., a product recall, conflict of medicines). In applications of this idiom, an interface of the form *ReportNewDatum()* is exported by the user to the service; the service invokes this interface upon arrival of a new datum, and the preserver is responsible for notifying the user of any matches or initiating corresponding actions. Notifications are encrypted and the preserver can batch them up, push them to the user or wait for pulls by the user, and even put in fake matches. Other examples include stock-trading, Google News Alerts etc.

*Analytics on Sensitive Data Idiom:* This idiom is characterized by expensive computations on large, sensitive user datasets. The provider executes a public algorithm on large datasets of a single user or multiple users, returning results back to the user, or using the results to offer a particular targeted service. Examples include targeted advertising and recommendation services. In targeted advertising, an activity log of a set of users (e.g., web visit logs, search logs, location trajectories) are mined to construct a prediction model. A slim interface for this scenario is a call of the form *SelectAds(ListOfAds)* which the preserver implements via statistical analysis.

*Proxying Idiom:* This idiom captures functionality whose purpose is to obtain the help of an external service. The user data in this case has the flavor of a capability for accessing that external service. For instance, consider the case when a user hands her credit card number (CCN) to a web service. If we restructure this application, an interface of the form *Charge(Amount, MerchantAccount)* would suffice; the preserver is responsible for the actual charging, and returning the confirmation code to the merchant. Other examples include confiding email addresses to websites, granting Facebook access to Gmail username/password to retrieve contacts, etc.

*Data Hosting Non-idiom:* For the idioms we discussed, the interface to user data is simple and narrow. To better characterize the scope of our work, we describe here a class of applications that do not fit within our work. Such applications rely on the service reading and updating the user's data at the finest granularity. Examples are collaborative document-editing services (e.g., Google Docs) or social networking services (e.g., Facebook itself, as opposed to Facebook Apps, which we described above). Applications in the hosting idiom are better handled by data-centric—instead of interface-centric—control mechanisms such as information flow control (e.g., XBook [27]) or end-to-end encryption (e.g., NOYB [14]).

### 2.5 Design Goals

Based on the applications we have discussed so far, we present our basic design goals:

**Simple Interface:** A user's data should be accessed only via a *simple, narrow interface*. The use of such interfaces is the key for allowing the user the flexibility to choose code from any provider that implements an interface, while offering confidence that said code is correct.

**Flexible Deployment:** *Flexible interposition* of the boundary between the user's data and the web service is necessary so that users and services can choose a suitable deployment option to match isolation requirements, at a given performance and cost budget.

**Fine-grained Use Policy:** Even given an interface, different uses may imply different permissions, with different restrictions. For example, a user may wish to use the Google Checkout service but impose restrictions on time-to-use and budget. As a result, we aim to allow *fine-grained and flexible user control* over how the interface to a user's data is exercised.

**Trust But Mitigate Risk:** Enforced interfaces are fine, but even assumptions on trusted mechanisms occasionally fail—in fact, we would not be discussing this topic without this painful truth. As a result, *risk mitigation* on what may be exposed even if a trusted encapsulation of a user's data were to be breached is significant. This is particularly important for multi-service workflows, where typically only a small subset of the raw user data or an anonymized form of raw user data need be provided to a subordinate service. For example, consider an advertising service that has access to all of a user's purchase history; if that service delegates the task of understanding the user's music tastes to a music recommendation service, there is no reason to provide the recommendation service with *all* history, but only music-related history.

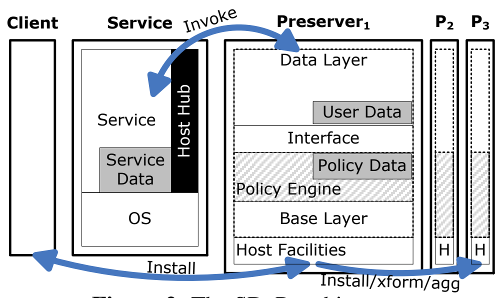

**Figure 2:** The SDaP architecture.

## 3 Preserver Architecture

In this section, we present the software components of the architecture, followed by an operational overview that shows their interaction.

### 3.1 Components

Figure 2 shows a preserver ($Preserver_1$) that serves a user's data to the service via a chosen interface; $P_2, P_3$ are preservers derived from $Preserver_1$ through hosting transfers or transformations. The three main components of a preserver are shown within $Preserver_1$. The *data layer* is the data-specific code that implements the required interface. The *policy engine* vets interface invocations per the user's policy stored alongside the user's data as *policy data*. Finally, the *base layer* coordinates these two components, and is responsible for: (a) interfacing with the service, (b) implementing the hosting protocol, and (c) invoking any required data transformations alongside hosting transfers. The base layer relies on *host facilities* for its interactions with the external world (such as network access). At the service side, the *host hub* module runs alongside the service. It serves as a proxy between the service code and the preserver, and allows us to decouple the particular application from the details of interactions with the preserver.

### 3.2 Operational View

The lifecycle of a preserver consists of installation, followed by any combination of invocations, hosting transfers, and transformations. We give an overview of these; our design section discusses them in detail.

A user who wishes to use preservers to protect her data when stored at a service, picks a preserver implementation from third party security companies (e.g., Symantec) that exports an interface suitable for that kind of data. Or, she may pick one from an open-source repository of preserver implementations, or purchase one from an online app-store. Another option is for the service itself to offer a third-party audited preserver which the user may trust. For such a rich ecosystem of preserver implementations, we envision that APIs suitable for specific kinds of data will eventually be well-defined for commonly used sensitive information such as credit card numbers (CCNs), email addresses, web histories, and trading strategies. Services that require use of a specific kind of data can support such an API, which can then be implemented by open-source reference implementations and security companies. The evolution of APIs like OpenSocial [3] are encouraging in this regard. Once the user picks a preserver implementation, she customizes it with policies that limit where her data may be stored and who may invoke it. We envision that users will use visual interfaces for this purpose, which would then be translated to our declarative policy language. Once customization is done, the user initiates an installation process by which the preserver is hosted (at TTP/client/service as desired) and the association between the service and the preserver established.

We discussed needed changes from the user's perspective; we now turn to the service's side. First, a service has to modify its code to interact with the preserver via a programmatic interface, instead of accessing raw data. Since web service code is rewritten much more frequently than desktop applications and is usually modular, we believe this is feasible. For instance, in the case of our charging preserver, the required service-side modifications took us only a day. Second, services need to run third-party preserver code for certain deployment options (colocation, preserver hosted on isolated physical infrastructure within the service). We believe this does not present serious security concerns since preserver functionality is very limited; preservers can be sandboxed with simple policies (e.g., allow network access to only the payment gateway, allow no disk access). A service can also insist that the preserver be signed from a set of well-known security companies. The overhead of preserver storage and invocation may also hinder adoption by services. Our evaluation section (Section 6) measures and discusses the implications of this overhead; this overhead amounts to the *cost of data isolation* in our framework.

Once the association between a service and a preserver is established, the service can make invocation requests via its host hub; these requests are dispatched to the base layer, which invokes the policy engine to determine whether the invocation should be allowed or not. If allowed, the invocation is dispatched to the data layer and the result returned. A service makes a hosting transfer request in a similar fashion; if the policy engine permits the transfer, the base layer initiates a hosting transfer protocol and initiates any policy-specified transformations alongside the transfer.

**Interaction between Base Layer and Host Hub:** The mechanics of interaction between the host hub (running alongside the service code) and the base layer (at the preserver) depends on the deployment scenario. If the preserver is hosted at a TTP or at the client, then this communication is over the network, and trust is guaranteed by the physical isolation between the preserver and the service. If the preserver is co-located at the service, then communication is achieved via the trusted module at the service site (e.g., Xen hypercalls, TPM late launch, secure co-processor invocation). The base layer requires the following functionality from such a trusted module: (a) isolation, (b) non-volatile storage, which can be used for anti-replay protection, (c) random number generation, and (d) remote attestation (we will argue in Section 6.3 that these four properties suffice for the security of our framework; for now, we note they are provided by all three trust modules we consider).

Details about base layers, policy engines, and aggregation modules follow in the next section; here we focus on the two roles of the host hub. The first is to serve as a proxy for communication to/from the preserver from/to the service. The second role applies only to colocation; it provides host facilities, such as network access, to the preserver. This lets the preserver leverage such functionality without having to implement it, considerably simplifying the implementation. The host hub runs within the untrusted service, but this does not violate our trust, since data can be encrypted if desired before any network communication via the host hub. The host hub currently provides three services: network access, persistent storage, and access to current time from a trusted remote time server (useful for time-bound policies).

## 4 Design

This section presents the design of two key components of the preserver: the policy engine and the data transformation mechanisms. We discuss the policy engine in two parts: the hosting policy engine (Section 4.1) and the invocation policy engine (Section 4.2). We then discuss data transformations (Section 4.3).

The main challenge in designing the policy layer is to design a policy language that captures typical kinds of constraints (based on our application scenarios), whilst enabling a secure implementation. We make the design decision of using a declarative policy language that precisely defines the set of allowed invocations. Our language is based on a simplified version of SecPAL [7] (a declarative authorization policy language for distributed systems); we reduced SecPAL's features to make simpler its interpreter, since it is part of our TCB.

### 4.1 Preserver Hosting

We introduce some notation before discussing our hosting protocol. A preserver $C$ storing user $U$'s data and resident on a machine $M$ owned by principal $S$ is denoted as $C_U@[M,S]$; the machine $M$ is included in the notation since whether $C$ can be hosted at $M$ can depend on whether $M$ has a trusted hardware/software module. Once a preserver $C$ (we omit $U,S,M$ when the preserver referred to is clear from the context) has been associated with $S$, it will respond to invocation requests per any configured policies (denoted by $P(C)$; policies are sent along with the preserver during installation). $S$ can also request hosting transfers from $C$'s current machine to one belonging to another service (for inter-organizational transfers) or to one belonging to $S$ (say, for replication). If this transfer is concordant with the preserver's *hosting policy*, then the *hosting protocol* is initiated; we discuss these next.

**Hosting Policy:** We wish to design a language flexible enough to allow a user to grant a service $S$ the right to host her data based on: (a) white-lists of services, (b) trusted hardware/software modules available on that service, and (c) delegating the decision to a service. We show an example below to illustrate the flavor of our language.

(1) `alice` SAYS CanHost(`M`) IF OwnsMachine(`amazon`,`M`)
(2) `alice` SAYS CanHost(`M`) IF TrustedService(`S`), OwnsMachine(`S`,`M`), HasCoprocessor(`M`)
(3) `alice` SAYS `amazon` CANSAY TrustedService(`S`)

We use a lower-case typewriter font to indicate principals like Alice (the user; a principal's identity is established by a list of public keys sent by the user or by a certificate binding its name to a public key), capitals to indicate language keywords, and mixed-case for indicating predicates. A predicate can either be built-in or user-defined. The predicate OwnsMachine*(S,M)* is a built-in predicate that is true if $S$ acknowledges $M$ as its own (using a certificate). The predicates HasCoprocessor*(M)*, HasTPM*(M)*, HasVMM*(M)* are built-in predicates that are true if $M$ can prove that it has a co-processor / TPM / VMM respectively using certificates and suitable attestations. The user-defined predicate CanHost*(M)* evaluates to true if machine $M$ can host the user's preserver. The user-defined predicate TrustedService*(S)* is a helper predicate.

The first rule says that `alice` allows any machine $M$ to host her preserver, provided `amazon` certifies such a machine. The second rule indicates `alice` allows machine $M$ to host her preserver if (a) $S$ is a trusted service, (b) $S$ asserts that $M$ is its machine, and (c) $M$ has a secure co-processor. The third rule allows `amazon` to recommend any service $S$ as a trusted service.

A hosting request received at $C_U@[M,S]$ has three parameters: the machine $M'$ to which the transfer is requested, the service $S'$ owning machine $M'$, and a set of assertions $P_{HR}$. The set of assertions $P_{HR}$ are presented by the principal in support of its request; it may include delegation assertions (such as "`S` SAYS `S'` CanHost*(X)*") and capability assertions (such as "`CA` SAYS HasTPM*(M')*"). When such a request is received by $C_U@[M,S]$, it is checked against its policy. This involves checking whether the fact "`U` SAYS CanHost*(M')*" is derivable from $P(C) \cup P_{HR}$ per SecPAL's inference rules. If so, then the hosting protocol is initiated.

**Hosting Protocol:** The hosting protocol forwards a preserver from one machine $M$ to another $M'$. The transfer of a preserver to a TTP is done over SSL; for colocation, the transfer is more complex since the protocol should first verify the presence of the trusted module on the service side. We rely on the attestation functionality of the trusted module in order to do so.

The goal of the hosting protocol is to maintain the confidentiality of the preserver during its transfer to the service side, whilst verifying the presence of the trusted module. We achieve this by extending the standard Diffie-Hellman key-exchange protocol:

- Step 1: $M \to M'$: $(g,p,A), N$
- Step 2: $M' \to M$: $B$, *Attestation*[ $M'$, BaseLayer, $N, (g,p,A), B$ ]
- Step 3: $M \to M'$: $\{C_U@[M,S]\}_{DHK}$

Here, $(g,p,A = g^a \ mod\ p)$ and $B = g^b \ mod\ p$ are from the standard Diffie-Hellman protocol, and $DHK$ is the Diffie-Hellman key (generated as $A^b \ mod\ p = B^a \ mod\ p$). This protocol only adds two fields to the standard protocol: the nonce $N$ and the attestation (say, from a Trusted Platform Module) that proves that the base layer generated the value $B$ in response to $((g,p,A),N)$. Thus, the security properties of the original protocol still apply. The attestation is made with an attestation identity key $M'$; this key is vouched for by a certification authority as belonging to a trusted module (e.g., TPM). The attestation guarantees freshness (since it is bound to $N$), and rules out tampering on both the input and output.

At the completion of the exchange, the base layer at $M'$ de-serializes the transferred preserver $C'_U@[M',S']$. At this point, the preserver $C'$ is operational; it shares the same data as $C$ and is owned by the same user $U$. After the transfer, we do not enforce any data consistency between $C$ and $C'$ (much like the case today; if Amazon hands out a user's purchase history to a third party, it need not update it automatically). A user or service can notify any such derived preservers and update them, but this is not automatic since synchronous data updates are typically not required across multiple web services.

## 4.2 Preserver Invocation Policy

An invocation policy allows the user to specify constraints on the invocation parameters to the preserver interface. Our policy language supports two kinds of constraints: stateless and stateful.

Stateless constraints specify conditions that must be satisfied by arguments to a single invocation of a preserver, e.g., "never charge more than 100 dollars in a single invocation". We support predicates based on comparison operations, along with any conjunction or disjunction operations. Stateful constraints apply across several invocations; for example, "no more than 100 dollars during the lifetime of the preserver"; such constraints are useful for specifying cumulative constraints. For instance, users can specify a CCN budget over a time window. We present an excerpt below.

```
(1) alice SAYS CanInvoke(amazon, A) IF LessThan(A,50)
(2) alice SAYS CanInvoke(doubleclick, A)
IF LessThan(A,Limit), Between(Time, "01/01/10","01/31/10")
STATE (Limit = 50, Update(Limit, A))
(3) alice SAYS amazon CANSAY CanInvoke(S,A)
IF LessThan(A,Limit)
STATE  (Limit = 50, Update(Limit, A))
```

The first rule gives the capability "can invoke up to amount A" to Amazon as long as $A < 50$. The second rule shows a stateful example; the semantics of this rule is that DoubleClick is allowed to charge up to a cumulative limit of 50 during Jan 2010. The syntax for a stateful policy is to annotate state variables with the STATE keyword. This policy has a state variable called *Limit* set to 50 initially. The predicate Update*(Limit,A)* is a built-in update predicate that indicates if this rule is matched, then the *Limit* should be updated with the amount $A$. When a rule is matched with a *state* keyword, it is removed from the policy database, any state variables (e.g., *Limit*) suitably updated, and the new rule inserted into the database. This usage idiom is similar to SecPAL's support for RBAC dynamic sessions. The alternative is to move this state outside the SecPAL policy, and house it within the preserver functionality; we avoid this so that the policy implementation is not split across SecPAL and the preserver implementation. The third rule is very similar to the second rule; however, this rule is matched for any principal to which Amazon has bestowed invocation rights. This means that the limit is enforced across all those invocations; this is exactly the kind of behavior a user would expect.

**Transfer of Invocation Policies:** We now discuss how the invocation protocol interacts with the hosting protocol. During a hosting transfer initiated from $C_U@[M,S]$ to $C'_U@[M',S']$, $C$ should ensure that $C'$ has suitable policy assertions $P(C')$ so that the user's policy specified in $P(C)$ is not violated. To ensure this, any policies $P_{HR}$ specified by $S'$ during the hosting request are added to $P(C')$ to record the fact that $C'$ operates under that context. Second, any stateful policies need to be specially handled, e.g., consider our third invocation policy: the total budget across all third parties that are vouched for by Amazon is 100 dollars. If this constraint is to hold across both $C'$ and any future $C''$ that might be derived from $C$, then one option is to use $C$ as a common point during invocation to ensure that this constraint is never violated. However, this requires any transferred preserver $C'$ to communicate with $C$ upon invocations. This is undesirable, and further, such synchronization is not required in most web service applications. Instead, we leverage the concept of exo-leasing [26]. *Decomposable constraints* (such as budgets, number of queries answered) from $P(C)$ are split into two sub-constraints; the original constraint in $P(C)$ is updated with the first, and the second is added to $P(C')$. For instance, a budget is split between the current preserver and the transferred preserver. We currently only support additive constraints which can be split in any user-desired ratio; other kinds of constraints can be added if required.

## 4.3 Preserver Data Transformation

This section discusses how to provide users control over data transformations. This is different from providing invocation control; the latter controls operations invoked over the data, and the former controls the data itself. We refer to a preserver whose data is derived from a set of existing preservers as a derivative preserver. We support two data transformations towards *aiding risk mitigation*: filtering and aggregation.

**Filtering:** A derivative preserver obtained by filtering has a subset of the original data; for instance, only the web history in the last six months. A preserver that supports such transformations on its data exports an interface call for this purpose; this is invoked alongside a hosting protocol request so that the forwarded preserver contains a subset of the originating preserver.

**Aggregation:** This allows the merging of raw data from mutually trusting users of a service, so that the service can use the aggregated raw data, while the users still obtain some privacy guarantees due to aggregation. A trusted aggregator preserver can also improve efficiency in the sensitive query idiom, since it enables a (private) index across preservers, sparing them from irrelevant events. We refer to the set of aggregated users as "data crowds" (inspired by the Crowds anonymity system [23]). We describe what user actions are necessary for enabling aggregation, and then discuss how the service carries it out.

To enable aggregation, a user $U$ instructs her preserver $C_U@[M,S]$ to aggregate her data with a set of preservers

$C_{U'}[M', S]$ where $U'$ is a set of users that she trusts. The set of users $U'$ form a data crowd. We envision that a user $U$ can discover such a large enough set of such users $U'$ by mining her social network (for instance). During preserver installation, each member $U$ of the crowd $C$ confides a key $K_C$ shared among all members in the crowd to their preserver. During installation, a user $U \in C$ also notifies the service of her willingness to be aggregated in a data crowd identified by $H(K_A)$ ($H$ is a hash function). $S$ can then identify the set of preservers $C_A$ belonging to that particular data crowd using $H(K_A)$ as an identifier.

To aggregate, the service expects the preserver to support an aggregation interface call. This call requests the preserver $C_U$ to merge with $C_{U'}$ and is a simple pair-wise operation. These mergers are appropriately staged by the service that initiates the aggregation. During the aggregation operation of $C_U$ with $C_{U'}$, preserver $C_U$ simply encrypts its sensitive data using the shared key $K_A$ and hands it off to $U'$ along with its owner's key. During this aggregation, the resultant derivative preserver also maintains a list of all preservers merged into the aggregate so far; preservers are identified by the public key of the owner sent during installation. This list is required so as to prevent duplicate aggregation; such duplicate aggregation can reduce the privacy guarantees. Once the count of source preservers in an aggregated preserver exceeds a user-specified constraint, the aggregate preserver can then reveal its data to the service $S$. This scheme places the bulk of the aggregation functionality upon the service giving it freedom to optimize the aggregation.

## 5 Implementation

Our implementation supports three deployments: TTP, client-side, and Xen-based colocation. We plan to support TPMs and secure co-processors using standard implementation techniques such as late launch (e.g., Flicker [18]). We implement three preservers, one per idiom: a stock trading preserver, a targeted ads preserver, and a CCN-based charging preserver (we only describe the first two here due to space constraints). We first describe our framework, and then these preservers (details are presented in our technical report [16]).

**Preserver Framework:** For TTP deployment, the network isolates the preserver from the service. The colocation deployment relies on Xen. We ported and extended XenSocket [34] to our setup (Linux 2.6.29-2 / Xen 3.4-1) in order to provide fast two-way communication between the web service VM and the preserver VM using shared memory and event channels. The base layer implements policy checking by converting policies to DataLog clauses, and answers queries by a simple top-down resolution algorithm (described in SecPAL [7]; we could not use their implementation since it required .NET libraries). We use the PolarSSL library for embedded systems for light-weight cryptographic functionality, since it is a significantly smaller code base (12K) compared to OpenSSL (over 200K lines of code); this design decision can be revisited if required. We use a TPM, if available, to verify that a remote machine is running Xen using the Trousers library. Note that we only use a TPM to verify the execution of Xen; we still assume that Xen isolates correctly.

**Stock Trading Preserver:** We model our stock preserver after a feature in a popular day trading software, Sierra Chart [4] that makes trades automatically when an incoming stream of ticker data matches certain conditions. This preserver belongs to the query idiom and exports a single function call *TickerEvent (SYMBOL, PRICE)* that returns an *ORDER("NONE" /"BUY" / "SELL", SYMBOL, QUANTITY)* indicating whether the preserver wishes to make a trade and of what quantity. The preserver allows the user to specify two conditions (which can be arbitrarily nested boolean predicates with operations like AND, OR, and NOT, and base predicates that consist of the current price, its moving average, current position, and comparison operations): a "BUY" and a "SELL" condition. Our implementation of predicate matching is straightforward; we apply no query optimizations, so our results are only meaningful for comparison.

**Targeted Advertising Preserver:** We implemented two preservers for targeted advertising (serving targeted ads and building long-term models), both in the analytics idiom. They store the user's browsing history and are updated periodically (say, daily).

The first preserver is used for targeted advertising for which we implemented two possible interfaces: *ChooseAd(List of Ads, Ad Categories)* and *GetInterestVector()*. In the first, the preserver selects the ad to be displayed using a procedure followed by web services today (described in Adnostic [29]). In the second, the preserver extracts the user's interest vector from her browsing history, and then perturbs it using differential privacy techniques (details are in our technical report [16]). This preserver uses a stateful policy to record the number of queries made (since information leak in interactive privacy mechanisms increases linearly with the number of queries). The second preserver allows the service to use any proprietary algorithm in serving ads since the feature vector, which summarizes a user's detailed history, is itself revealed after being appropriately perturbed with noise. This preserver is used by the service to build long-term models related to the affinity of users with specific profiles to specific advertisements; our aggregation functionality is of use here.

Based on these preservers, we estimate the typical refactoring effort for a web service in adopting the preserver architecture. We do not have access to a stock broker's code base, so our experience is based on the targeted ads preserver and the CCN case. For the targeted ads case, given the availability of client-side plugins like Adnostic and PrivAd, we believe the refactoring effort will be minimal. In the CCN case, modifying the Zen shopping cart [6] to interact with the preserver took us only a day.

## 6 Evaluation

In this section, we first present the performance evaluation of our framework, followed by a security analysis. Our performance evaluation has two goals. First, evaluate *the cost of isolation in our framework* as the performance overhead and provisioning cost of client-side / TTP / colocation deployment, relative to the base case (no isolation). Second compare *various deployment options* to determine the ideal deployment for different workloads.

**Experimental Scenarios:** We consider three scenarios: (a) the base case, (b) the TTP case, and (c) the Xen-based colocation case. Since a TTP deployment is equivalent in most respects to a client-side preserver, we discuss them together. We compare these scenarios along three dimensions of performance: setup cost, invocation cost, data transformation cost. The term *cost* includes latency, network bandwidth consumption, and storage cost. Of these, the invocation cost is borne every time the user's data is accessed, and is thus the primary metric of comparison. The setup cost is incurred only during the initial and subsequent transfers of the user's data (since this is an infrequent event, the details are omitted from this paper; they are in our technical report [16]), while the transformation cost, though not as frequent as invocation, may be significant if the aggregation involves data belonging to large numbers of users. All results are reported over 100 runs. For latency, we report the median and the 95% confidence interval for the median; we report only the median for bandwidth since it is much less variable.

**Hardware Configuration:** Our test server is a 2.67 GHz quad core Intel Xeon with 5 GB memory. A desktop (Intel Pentium4 1.8 GHz processor, 1 GB memory), on the same LAN as the server, served as a TTP (or client). The bandwidth between the server and this desktop was 10 Gbps and the round-trip about 0.2 ms. To simulate a wide-area network between the client/TTP and the service, we used DummyNet [2] to artificially delay packets by a configurable parameter; the default round-trip is 10 ms.

### 6.1 The Cost of Isolation

We measured the performance metrics during invocation and setup, and then examine provisioning requirements. For these measurements, we used a dummy data layer that accepts an invocation payload of a specific size and returns a response of a specific size.

**Invocation Costs:** To examine the invocation cost across different payload sizes, we plotted the latency as a function of payload size (varied in multiples of 2 from 256 bytes to 32 KB) in Figure 3(left). At 1 KB invocation size, though the latency via Xen (1237$\mu$s) is about 800 $\mu$s worse compared to the base case (415$\mu$s), it is still significantly lower compared to the TTP case (24.37ms). We found considerable variation ranging from 900 to 4000$\mu$s with a median of 1237$\mu$s; we believe this is related to the Xen scheduling algorithm which may delay the execution of the preserver VM. This plot shows that the overhead added by Xen as a percentage of the base case declines, while the network transfer time increases the latency for the TTP case. In the TTP case, the latency is due to two round-trips (one each for exchange of TCP SYN and SYN ACKs, and the invocation), and network transfer time. The traffic for the TTP case varies roughly linearly with the size of the payload: 1.7 KB (for invocations of 256 bytes), 10 KB (4 KB), and 72 KB (32 KB); of course, in the Xen case and the base case, no network communication is involved for invocation.

**Provisioning Cost:** We first estimate the colocation provisioning cost and the monetary/provisioning costs for TTP/client-side preservers.

Under colocation, preserver storage requires memory and disk; our preserver code is about 400 KB (120 KB compressed) from 100 SLOC. The overhead is due to glibc which we plan to remove. We used the Difference Engine [15] which finds similar pages across VMs to reduce memory costs; thus, the memory requirements for the VMs of users who share the preserver code from the same provider are significantly reduced. In our experiments, we initially allocated 64 MB of memory to each preserver VM, and then invoked a varying number of client VMs with the same preserver. The Difference Engine saves us about 85% overall memory; the memory requirement per preserver VM is about 10 MB (this estimate was obtained by invoking 10 VMs, and then allowing time for the detection of identical and similar memory pages). At the current estimate of 10 MB per preserver, every 100 users require about 1 GB memory. We believe that this can be reduced further since we do not use any kernel facilities; thus, the preserver can run directly on the VMM.

The trade-offs are different for TTP/client-side preservers. The TTP shoulders the cost of availability and performance. Web-service hosting (with unlimited bandwidth) are about $10 - 15$ dollars per month today; expectations are higher from a *trusted* service because of the security risks, so one may expect a higher cost. In the client-side preserver, availability and performance fall on the user instead.

### 6.2 Performance of Various Preserver Deployments

We evaluated the stock trading preserver and targeted ads preserver for comparing the deployments since they represent different workloads (frequent small invocation versus low-frequency invocation with larger payload). The CCN preserver's results are in our technical report [16].

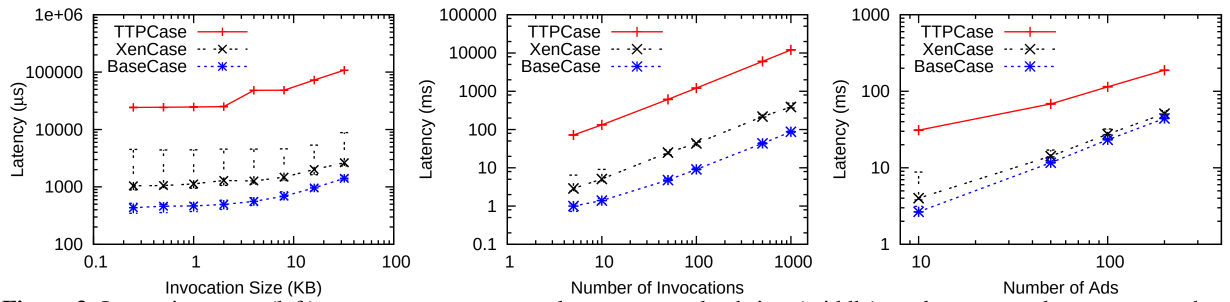

**Figure 3:** Invocation costs: (left) measurement preserver; latency vs. payload size, (middle) stock preserver; latency vs. number of ticker events, (right) targeted ads preservers; latency vs number of ads.

**Stock Preserver:**To reflect a typical trading scenario with hundreds of ticker events per second, we plotted latency versus numbers of back-to-back invocations (stock ticker events) in Figure 3(middle). As expected, the TTP case requiring network access is substantially slower as compared to the Xen case and the base case. Comparing the Xen case and the base case for a sequence of 100 back-to-back invocations, the latencies are 43.4ms and 9.05ms respectively; though the overhead due to Xen is substantial, it is still a significant improvement over the TTP case, which requires 12 seconds. Regarding the traffic, in the TTP case, the traffic generated for a sequence of $5, 10, 50, 100, 500$ and $1000$ invocations are respectively $2.8, 4.8, 20.2, 39.4, 194, 387$ KB respectively. Thus, the bandwidth required for say, 500 events per second, is about 1.6 MB/s per user per stock symbol (and a single user typically trades multiple symbols). These results reflect that server site colocation can offer significant benefit in terms of network bandwidth.

**Targeted Ads Preserver:** We present two results for the targeted ads scenario. Figure 3(right) shows the latency per invocation for the preserver to serve targeted ads using the *ChooseAd* interface (we do not show results for the *GetInterestVector* interface due to lack of space). This graph shows that the Xen preserver has nearly the same overhead as the Base case preserver once the number of ads out of which the preserver selects one exceeds 5. This is mainly because the payload size dominates the transfer time; the context switch overhead is minimal. In the TTP case, the latency is clearly impacted by the wide-area network delay. The traffic generated in the TTP case was measured to be $2.6, 8.3, 15.8, 29.2$ KB for $10, 50, 100, 200$ ads respectively. These reflect the bandwidth savings of colocation; since this bandwidth is incurred for every website the user visits, this could be significant. Our technical report [16] has performance results on two other operations: allowing a user to update her web history and the data aggregation transformation.

## 6.3 Security Analysis

We now discuss the desirable security properties of our preserver, determine the TCB for these to hold, and argue that our TCB is a significant improvement over the status quo. Our security goal is that any data access or preserver transfer obeys the user-sanctioned interface and policies. Depending on the deployment, the adversary either has control of the service software stack (above the VMM layer) or has physical access to the machine hosting the preserver; the first applies to VMMs, TPMs, and the second to TTPs, client-side preservers, secure co-processors. In the TTP/client case, we rely on physical isolation and the preserver implementation's correctness; the rest of this section is concerned with the security goal for colocation.

Colocation has one basic caveat: the service can launch an availability attack by refusing to service requests using its host hub or by preventing a user from updating her preserver. We assume that the trusted module provides the four security properties detailed in Section 3: isolation, anti-replay protection, random number generation, and remote attestation. We also assume that any side-channels (such as memory page caching, CPU usage) have limited bandwidth (a subject of ongoing research [30]). We first analyze the installation protocol and then examine the invocation protocol.

**Installation Protocol:** The installation protocol is somewhat complex since it involves a Diffie-Hellman key exchange along with remote attestation. In order to argue its correctness, we now present an informal argument (our technical report [16] has a formal argument in $LS^2$ [10], a formal specification language). Our informal argument is based on three salient features of the installation protocol (Section 4.1). First, the attestation guarantees that the public key is generated by the base layer. Second, the Diffie-Hellman-based installation protocol ensures confidentiality; only the party that generated the public key itself can decrypt the preserver. From the first observation, this party can only be the base layer. This relies on the random number generation ability of the trusted module to ensure that the generated public key is truly random. Third, since attestation verifies the purported input and output to the code which generated the public key, man-in-the-middle attacks are ruled out; an adversary cannot tamper with the attestation without rendering it invalid. These three properties together ensure that the preserver can only be decrypted by the base layer, thus ruling out any leakage during installation. The correctness of the base layer's implementation helps argue that the preserver is correctly decrypted and instantiated at the service.

**Invocation Protocol:** Upon invocation, the base layer verifies: (a) the invoking principal's identity, (b) any supporting policies, and (c) that user-specified policies along with supporting policies allow the principal the privilege to make the particular invocation. The principal's identity is verified either against a list of public keys sent during installation (which binds service names to their public keys or offloads trust to a certification authority). In either case, the correctness of installation ensures that the identity cannot be spoofed. The base layer verifies the supporting policies by verifying each statement of the form $X\ SAYS\ \cdots$ against the signature of $X$. The policy resolver takes two sets of policies as input: user-specified policies and the invoker's supporting policies. The latter, we have argued, is correct; for the former, we rely on the anti-replay property provided by the trusted module to ensure that the preserver's policies (which can be updated over time) are up-to-date and reflects the outcome of all past invocations. This ensures that any stateful constraints are correctly ensured. Once invocation is permitted by the base layer, it is passed on to the data layer which implements the interface. In cases such as a query preserver or an analytics preserver, this functionality is carried out entirely within the data layer, which we assume to be correct. For the proxy preserver, which requires network access, we note that though the network stack is itself offloaded to the host hub, the SSL library resides in the preserver; thus the host hub cannot compromise confidentiality or integrity of network communication.

### 6.3.1 TCB Estimate

From the preceding discussion, it is clear that our TCB includes: (A) the trust module, (B) the data layer interface and implementation, and (C) the base layer protocols and implementation. In our colocation implementation, (A) is the Xen VMM and Dom0 kernel; we share these with other VMM-based security architectures (e.g., Terra [13]). Mechanisms to improve VMM-based security (e.g., disaggregation [19] removes Dom0 from the TCB) also apply to our framework. Regarding the base layer and the data layer, their per-module LOC estimates are: Base Layer (6K), PolarSSL (12K), XenSocket (1K), Trousers (10K), Data Layers (Stock Preserver: 340, Ads: 341, CCN: 353). This list omits two implementation dependencies we plan to remove. First, we boot up the preserver atop Linux 2.6.29-2; however, our preservers do not utilize any OS functionality (since device-drivers, network stack, etc., are provided by the host hub), and can be ported to run directly atop Xen or MiniOS (a barebones OS distributed with Xen). Second, the preservers use *glibc*'s memory allocation and a few STL data structures; we plan to hand-implement a custom memory allocator to avoid these dependencies. We base our trust in the data layer interface and implementation in the interface's simplicity. Currently, despite our efforts at a simple design, the base layer is more complex than the data layer, as reflected in the LOC metric. In the lack of a formal argument for correctness, for now, our argument is that even our complex base layer offers a significant improvement in security to users who have no choice today but to rely on unaudited closed source service code.

## 7 Related Work

Before examining broader related work, we discuss three closely related papers: Wilhelm's thesis [31], CLAMP [21], and BStore [9]. Work in the mobile agent security literature, such as Wilhelm's thesis [31], leverages mobile agents (an agent is code passed around from one system to another to accomplish some functionality) to address data access control in a distributed system. Our main differences are: (a) our interface is data dependent and its invocation can be user-controlled, and (b) preservers return some function of the secret data as output; this output provides some useful secrecy guarantees to the user. CLAMP [21] rearchitects a web service to isolate various clients by refactoring two security-critical pieces into stand-alone modules: a query restrictor (which guards a database) and a dispatcher (which authenticates the user). Our goal is different: it is to protect an individual user's data (as opposed to a shared database), both from external and internal attacks. BStore [9] argues for the decoupling of the storage component from the rest of the web service: users entrust their files to a third party storage service which enforces policies on their behalf. Our preserver architecture is in a similar spirit, except that it pertains to *all* aspects of how data is handled by the web service, not just storage; the enforcement of an interface means that user data is never directly exposed to the web service. Our work is also related to the following areas:

**Information Flow Control (IFC):** The principle of IFC has been implemented in OSes (e.g., Asbestos [12]) and programming languages (e.g., JIF [24]), and enables policy-based control of information flow between security compartments. IFC has also been used to build secure web service frameworks, e.g., W5 [17], xBook [27]. Preservers provide data access control; this is complementary to IFC's data propagation control. Preservers rely on an interface that offers sufficient privacy to the user and is usable to the service. The advantage of interface-based access control is that we can rely on a variety of isolation mechanisms without requiring a particular OS or programming language. Further, the interface's simplicity makes it feasible to envision proving a preserver's correctness; doing so in the IFC case requires one to prove the correctness of the enforcement mechanism (OS or compiler) which can be significantly more complex.

**Decentralized Frameworks For Web Services:** Privacy frameworks that require only support from users have been proposed as an alternative to web services. VIS [8] maintains a social network in a completely decentralized fashion by users hosting their data on trusted parties of their own choice; there is no centralized web service. Preservers are more compatible with the current ecosystem of a web service storing users' data. NOYB [14] and LockR [28] are two recent proposals that use end-to-end encryption in social network services; both approaches are specific to social networks, and their mechanisms can be incorporated in the preserver framework, if so desired.

## 8 Conclusion

Our preserver framework rearchitects web services around the principle of giving users control over the code and policies affecting their data. This principle allows a user to decouple her data privacy from the services she uses. Our framework achieves this via *interface-based access control*, which applies to a variety of web services (though not all). Our *colocation* deployment model demonstrates that this property can be achieved with moderate overhead, while more stringent security can be obtained with other deployment models. In the future, we hope to formalize two aspects of our framework. First, we wish to prove the correctness of our interface implementation. Second, we hope to define and prove precise guarantees on information leak via interfaces.

**Acknowledgments:** We thank Gautam Altekar, Kevin Fall, Jon Howell, Eddie Kohler, Jay Lorch, Radia Perlman, and the anonymous reviewers for their careful feedback on our work, as well as the authors of Adnostic, in particular Arvind Narayanan, for sharing their data.

## References

[1] DataLoss DB: Open Security Foundation. `http://datalossdb.org`.

[2] DummyNet Homepage. `http://info.iet.unipi.it/~luigi/dummynet/`.

[3] OpenSocial. `http://code.google.com/apis/opensocial/`.

[4] Sierra Chart: Financial Market Charting and Trading Software. `http://sierrachart.com`.

[5] TPM Main Specification Level 2 Version 1.2, Revision 103 (Trusted Computing Group). `http://www.trustedcomputinggroup.org/resources/tpm_main_specification/`.

[6] Zen E-Commerce Solution. `http://www.zen-cart.com/`.

[7] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and Semantics of a Decentralized Authorization Language. In *Proc. IEEE Computer Security Foundations Symposium*, 2006.

[8] R. Cáceres, L. Cox, H. Lim, A. Shakimov, and A. Varshavsky. Virtual Individual Servers as Privacy-Preserving Proxies for Mobile Devices. In *Proc. MobiHeld*, 2009.

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

[10] A. Datta, J. Franklin, D. Garg, and D. Kaynar. A Logic of Secure Systems and its Application to Trusted Computing. In *IEEE Security and Privacy*, 2009.

[11] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, and S. Smith. Building the IBM 4758 Secure Coprocessor. *Computer*, 34(10), 2001.

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

[13] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *SOSP*, 2003.

[14] S. Guha, K. Tang, and P. Francis. NOYB: Privacy in Online Social Networks. In *Proc. WOSP*, 2008.

[15] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *OSDI*, 2008.

[16] J. Kannan, P. Maniatis, and B.-G. Chun. A Data Capsule Framework For Web Services: Providing Flexible Data Access Control To Users. arXiv:1002.0298v1 [cs.CR].

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

[18] J. M. Mccune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *EuroSys*, 2008.

[19] D. G. Murray, G. Milos, and S. Hand. Improving Xen Security through Disaggregation. In *VEE*, 2008.

[20] G. C. Necula. Proof-carrying code: Design, Implementation, and Applications. In *Proc. PPDP*, 2000.

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

[22] L. Ponemon. Fourth Annual US Cost of Data Breach Study. `http://www.ponemon.org/local/upload/fckjail/generalcontent/18/file/2008-2009 US Cost of Data Breach Report Final.pdf`, 2009. Retrieved Feb 2010.

[23] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security*, 1(1), 1998.

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

[25] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[26] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers. In *Middleware*, 2008.

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

[28] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better Privacy for Social Networks. In *CoNEXT*, 2009.

[29] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *NDSS*, 2010.

[30] E. Tromer, A. Chu, T. Ristenpart, S. Amarasinghe, R. L. Rivest, S. Savage, H. Shacham, and Q. Zhao. Architectural Attacks and their Mitigation by Binary Transformation. In *SOSP*, 2009.

[31] U. G. Wilhelm. *A Technical Approach to Privacy based on Mobile Agents Protected by Tamper-Resistant Hardware*. PhD thesis, Lausanne, 1999.

[32] B. S. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.

[33] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving Application Security with Data Flow Assertions. In *SOSP*, 2009.

[34] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. XenSocket: A High-Throughput Interdomain Transport for Virtual Machines. In *Middleware*, 2007.