Cloud-as-a-Gift: Effectively Exploiting Personal Cloud Free Accounts via REST APIs

Raúl Gracia-Tinedo, Marc Sánchez Artigas and Pedro García López
Universitat Rovira i Virgili, Tarragona (Spain)
{raul.gracia|marc.sanchez|pedro.garcia}@urv.cat

***Abstract*—Personal Clouds, such as DropBox and Box, provide *open REST APIs* for developers to create clever applications that make their service even more attractive. These APIs are a powerful abstraction that makes it possible for applications to transparently manage data from user accounts, blurring the lines between a Personal Cloud service and storage IaaS. Jointly, Personal Clouds also offer *free accounts* to lure new users, that normally include reduced storage space and unlimited transfers.**

However, the unintended consequence of combining open APIs and free accounts is that these companies are exposing *automated access to a free storage infrastructure*, which may lead to abuse by malicious parties. By exploiting the *freemium* API service, users may fraudulently consume resources or they can use free accounts as a Cloud storage layer to support abusive applications. We call this vulnerability the *storage leeching problem*.

In this paper, we show how easy it is to implement a *file-sharing application* able to distribute digital content by abusing Personal Clouds. Making use of open APIs, this application transparently aggregates the limited-space free accounts from multiple providers into a single larger storage layer, while achieving better transfer speed than that received from one provider alone. This demonstrates that free accounts can be easily exploited to obtain a practical Cloud storage service, and therefore, the potential impact of storage leeching.

***Index Terms*—Cloud Storage; Personal Clouds; Security**

I. Introduction

The enormous hype around the Personal Cloud model [1] has promoted the appearance of a myriad of very competitive offerings (DropBox, Box) that nowadays populates the market. This makes, in turn, Personal Clouds to aggressively react and improve their service to retain their market share. Essentially, Personal Clouds make their offering more interesting for new customers in two ways: adding innovative functionalities to their service and massively delivering *freemium* accounts.

Regarding the first point, Personal Clouds are incorporating a large corpus of value-added functionalities to their service (e.g. collaborative editors, media viewers). In this sense, major companies provide open REST APIs for developers to create clever applications that make their service even more attractive. From a functional viewpoint, these APIs enable an application to upload/download files to/from *user accounts*, blurring the lines between a Personal Cloud service and a pure IaaS provider as Amazon S3. Such a powerful abstraction hides the complexity of block-level data management and constitutes a rich substrate to cultivate a developer ecosystem.

Secondly, Personal Clouds offer *free accounts* to lure new customers and gain market share. These free accounts normally include reduced off-site storage space, as well as virtually unlimited transfers. Moreover, as paid accounts, free accounts provide standard functionalities, such as access from syncing software clients and Web front-ends. The impact of this *freemium* business model is remarkable: from the 50 million of DropBox users only the 4% pay for storage [2].

However, the unintended consequence of combining REST APIs and free accounts is that these companies are exposing *automated access to a free storage infrastructure*, which may lead to abuse by malicious parties. Nothing prevents a malicious user from acquiring an arbitrary number of free accounts from a single vendor and access to them via REST APIs, given the quick registration process that it requires. Furthermore, that user may aggregate accounts from various providers to build a larger and even better storage facility by exploiting storage diversity. Although aggregating free accounts might not be interpreted as an attack by itself, thanks to open APIs these accounts can be used to materialize illicit actions against Personal Clouds. For instance, users may perpetrate DDoS attacks, fraudulent resource consumption, or they could use free accounts as a storage layer to support abusive applications. We call this vulnerability the *storage leeching problem*.

The present paper describes the roots of the *storage leeching problem* and shows how easy is to benefit from it. To this end, we implemented `Boxleech`: a *proof-of-concept file-sharing application* able to distribute digital content by abusing Personal Clouds. This application transparently aggregates the limited-space free accounts from multiple providers into a single larger storage space while achieving better transfer performance than that received from a single provider. The feasibility of `Boxleech` yields that malicious users may *transform a set of Personal Cloud accounts into a powerful file-sharing system to distribute digital content*, being difficult to detect. Actually, users not registered in any Personal Cloud would only need to install the `Boxleech` client and acquire the meta-data file of the desired content to download it. This gives a notion about the potential extent of this kind of abuse.

Finally, we provide some discussion about possible countermeasures to deliver a more secure API service to developers.

The rest of the paper is organized as follows. We discuss the related work in Section II. We provide some background about Personal Cloud APIs in Section III. We describe the *storage leeching problem* in Section IV. The design of `Boxleech` appears in Section V. In Section VI we show the evaluation of `Boxleech` compared with Personal Cloud software clients. We discuss about possible countermeasures in Section VII and we conclude in Section VIII.

II. Related Work

In this work, we argue that Personal Clouds may be abused though their free account REST API service. For this reason, we found specially interesting recent efforts regarding security

in Web Services [3]. In this sense, the authors of [4] observe that the current lack of integrity controls at the data level in API REST Web Services could result in profound problems regarding data integrity. Other works such as [5] exploit specific vulnerabilities on the authentication mechanisms employed in Amazon EC2 and Eucalyptus Cloud control interfaces.

The abuse of Cloud services is currently a relevant research concern. As described in [6], one of the major risks of Cloud computing is its "abuse and nefarious use" by malicious parties (e.g. botnets, software exploits). In this line, few works analyzed the impact of external attacks on Cloud services and applications. For instance, authors in [7], [8] investigate the potential vulnerabilities of the Cloud computing model, which could be exploited from fraudulent resource consumption of any Internet connected host. Directly related with Personal Clouds, authors in [9] subvert the DropBox client to hide files in the Cloud with unlimited storage capacity.

We prove our findings by designing and evaluating an application capable to abuse Personal Clouds via REST APIs. Regarding abusive applications, few previous works presented systems which benefit from the available Internet services. Close to our work, EMFS [10] is a personal storage system which aggregates Cloud storage by establishing a RAID-like system on top of e-mail accounts. Other works propose backup tools or file systems benefiting from a variety of remote services, such as caches of Internet search engines, e-mail accounts and free web space [11], [12].

In contrast with previous research, the present work is the first to study the potential of Personal Cloud REST APIs as a building block for many abusive applications. Furthermore, we developed and evaluated a file-sharing application to show how easy it is to exploit these services.

III. Background

Personal Clouds provide open REST APIs, as well as their client implementations, to make it possible for developers the creation of novel applications which use the information stored in user accounts. In this section, we will describe the functioning of these APIs and the procedure needed to register an application to enable its access to user storage. Due to space constraints, we will describe the complete process for DropBox at the time of this writing.

Registering our application with DropBox. A Personal Cloud *application* is an authorized namespace within the Personal Cloud domains which enables REST API calls over user accounts. In DropBox, these applications are either in production or development states. The former means that the application has been revised and approved by DropBox, whereas the latter has limited features (development purposes). The DropBox API incorporates OAuth [13] authorization mechanism to manage the credentials/tokens of applications and users granting access to these applications. Note that with a DropBox application in development state, *a user is able to access up to* 5 *free storage accounts through the REST API*.

DropBox provides 3 subdomains to support its API service: i) `dropbox.com` corresponds to the webpage, the place where users and developers perform manual interactions (as explained later on), ii) `api.dropbox.com` is the subdomain against which applications perform authentication and metadata requests, and iii) `api-content.dropbox.com` is the subdomain where DropBox handles API data management operations (put, get). In the latter case, these operations are executed against Amazon S3, the storage backend of DropBox.

Now, we describe in general terms how to make an application operational in DropBox. We denote the application to be registered as A, DropBox as DB, and a user U that permits the access to his storage space. The procedure is as follows.

First, a developer registers A via DB's webpage (`dropbox.com` subdomain), where DB creates an *application token pair* that it will use to authenticate A. Second, A asks for a request token to DB. Note that A performs this step using DB's API, and therefore, addressing a request via a HTTP `POST` message to the `api.dropbox.com` subdomain. As a result, DB replies to A with a *request token pair*. Thirdly, U authorizes A via DB's webpage. Normally, U is redirected to DB's webpage by a link containing A's *request token* as argument. With this information, DB knows that user U is giving access to A. In fourth place, once U authorizes A, DB automatically notifies A about this event. DB generates the *access token* for A, which grants access to U's storage space. Next, A performs an API call to DB asking for the *access tokens*. Finally, A performs storage operations against U's account. The only requirement in each API call (get, put) is to include the *access token* in the request.

Once provided the necessary background to understand the functioning of these APIs and the procedure to register an application, we proceed to describe the *storage leeching problem*: the exploitation of REST API access to free accounts as mean to abuse Personal Clouds.

IV. The Storage Leeching Problem

Despite the trumpeted business and advantages of the Personal Cloud storage, many potential Cloud users have yet to join the Cloud. To make their offering even more attractive, major companies such as DropBox, Box and SugarSync, to name a few, provide open REST APIs for developers to create clever applications over their service. From a functional viewpoint, these APIs enable an application to upload/download files to/from user accounts, blurring the lines between a Personal Cloud service and a pure IaaS storage provider as Amazon S3. However, the unintended consequence is that it is very easy for a user to aggregate multiple free accounts from the same or from different Personal Clouds to obtain a free storage space comparable to paid accounts.

The roots of the problem lie deeply in the lack of *accountable* identities. Personal Clouds do not provide mechanisms to enforce the rule that *one real person* gets *one virtual identity* in their online services, what is known as the *Sybil attack* [14]. As an illustrative example, Box requires only the first name, last name, email and password for a user to set up an account of 5 GB of free storage. This *quick* registration process makes it possible for one real person to get multiple accounts and here is when the open nature of these REST APIs facilitate the abuse of the storage service. Box REST API allows a developer to enable up to four other users per application yet in development status, so nothing prevents a malicious developer from aggregating his 25 GB of free storage as a single unit. In

the case of Box, this new form of abuse may have economic consequences. At the time of this writing, a Box account of 25 GB costs $9.99 per month.

The extent of the abuse can be even worse if the abuser aggregates accounts from multiple providers. In such a case, the abuser can take benefit of storage diversity to obtain even a better service than what can be delivered from a single provider. By an intelligent allocation of file chunks to different providers, a malicious user can improve download times, upload times or both, and obtain a unified account with better QoS than a paid account totally free of charge.

We use the term "*storage leeching*" to refer to this generic form of abuse because the abusers or leechers seek to benefit from free storage while trying to leave unnoticed. This form of abuse is hard to prevent because it is under the umbrella of the *freemium* business model adopted by Personal Cloud companies. That is, storage providers offer free and paid premium accounts that are very similar in all aspects except for the amount of storage space offered. This, in conjunction with the business strategy to cultivate a developer ecosystem through the release of open APIs, makes it really hard for these companies to prevent storage leeching.

To illustrate the potential consequences of storage leeching, let us describe a real example. During the development of this piece of research, we executed several experiments against the REST API service of three major vendors: Box, DropBox and SugarSync. We consumed around 45.26TB of download traffic, 25.75TB of upload traffic and 450GB of storage. Excluding the number of transactions, in terms of Amazon S3 pricing[1], *our experiments represent a cost of* $\$5,431.2$ *in download traffic, plus a monthly storage cost of* $\$42.75$. This evidences that it is very easy to exploit these services.

We believe that the storage leeching problem is a substrate over which many abusive applications might exploit Personal Clouds. For instance, a single user may aggregate free accounts as a storage backend to support an *illegal webpage* which exhibits prohibited contents or even, as a part of a *peer-assisted storage* system [15]. Even worse, a malicious user may share with others the access tokens of a certain account, which enables any other user to access the stored data. The potential damage of this form of exploitation may be important, since it leverages the creation of applications such as *file-sharing*, where *users not registered in any Personal Cloud can freely consume resources and illicitly benefit from these services*.

In what follows, we argue about the potential threat that storage leeching represents for Personal Cloud vendors by estimating the hypothetical cost of this form of abuse.

A. Problem Motivation: Economic Impact of Storage Leeching

To motivate the relevance of the storage leeching problem, in this section we provide a simple cost model to give a sense about the economic impact of users abusing Personal Clouds.

The economic impact of users abusing a service is mainly given by two factors: the *number of users* and their *usage behavior*. We model these aspects as follows.

Simple cost model. We assume that new users arrive to the system and start using a certain abusive application that benefits from storage leeching. First, we consider discrete time intervals, denoted by $n \in \mathbb{N}$, of duration Δ. T represents time where the system is serving users. We denote by λ the average rate of new user arrivals per time interval. Analogously, we denote by μ the rate of users that permanently leave the system every time interval. Therefore, the total number of alive users abusing the system at time n is given by $N(n) = N(0) + n\lambda - n\mu$, where $\lambda \geq \mu$, and $N(0)$ represents the initial number of users which are already in the system.

Regarding storage, we consider that users are responsible for creating new storage accounts, as well as for gathering the necessary tokens to enable an application exploiting the service. Note that users have strong incentives for doing so, since that storage space will be exploited by themselves, or by the application they want to benefit from.

The fraction of users that creates storage accounts of size a when they arrive to the system is defined by $f_s \in [0, 1]$. Therefore, we do not consider that a user creates accounts over time. We also consider that once a user creates an account, he does not cancel it after he leaves the system. In this sense, as Personal Cloud accounts do not expire, the available storage to exploit will be always increasing. Thus, the maximum amount of available storage (S_a) at time n is:

$$S_a(n) = n \cdot \lambda \cdot f_s \cdot a, \tag{1}$$

To represent the actual consumption of storage resources (S_c), we employ an average of storage consumption per user, every time interval n, namely s. Note that the amount of consumed storage must be always $S_c(n) \leq S_a(n)$. Hence, the storage consumption in our model at time n is:

$$S_c(n) = \sum_{i=0}^{n} N(i) \cdot s, \tag{2}$$

The majority of abusive applications will consume download traffic, which at its turn, is an expensive utility of a Cloud storage service. The amount of download traffic consumed by users at time n is expressed as follows:

$$D(n) = \sum_{i=0}^{n} N(i) \cdot d, \tag{3}$$

where d denotes the average amount of consumed download traffic per time-slot n by every user. Note that, contrary to storage, Personal Cloud free accounts provide unlimited transfers, which has been confirmed in our experiments.

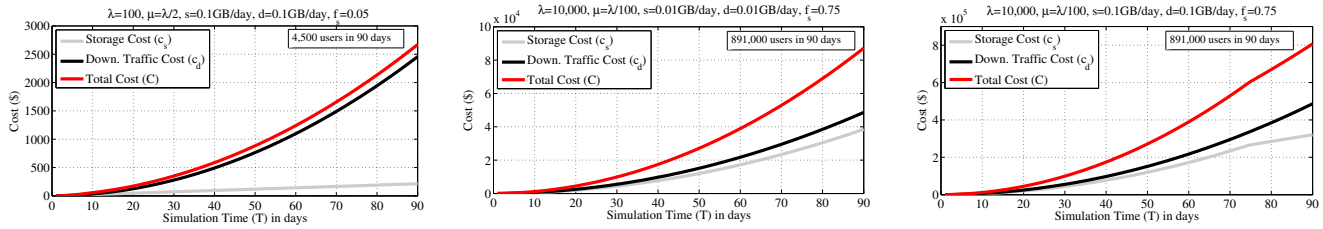
We have described how to estimate the amount of consumed resources at time n, in terms of storage and download traffic. To convert this into a monetary metric, we apply the appropriate pricing by unit of data as follows:

$$C(n) = S_c(n) \cdot c_s + D(n) \cdot c_d, \tag{4}$$

where c_s represents the monetary cost per storage unit and time interval, and c_d the price of downloading a unit of data.

We intentionally left upload traffic consumption out of our cost estimation model. The reason is that our objective is to estimate the monetary costs of storage leeching and none of the major Cloud providers (Amazon S3, Google Storage) charges any cost for the upload traffic.

[1] http://aws.amazon.com/en/s3/pricing/



(a) (b) (c)

Fig. 1. Cost estimation of storage leeching under various scenarios.

Param.	Description	Parametrization
λ	User arrivals per interval t	$100,\ 10,000$
μ	User departures per interval t	$\lambda/2,\ \lambda/100$
$N(0)$	Initial number of users	0
a	Account size	5GB
s	Storage consumption	0.01GB/day, 0.1GB/day
d	Down. traffic consumption	0.01GB/day, 0.1GB/day
f_s	Fraction of storage consumers	0.05, 0.75
c_s	Storage cost (Amazon S3)	0.095$/GB month
c_d	Down. traffic cost (Amazon S3)	0.12$/GB
Δ	Interval duration	1 day
T	Simulation time	90 days

TABLE I
COST MODEL PARAMETRIZATION USED IN OUR SIMULATIONS

Simulation results. Next, we analyze the hypothetical economic impact of storage leeching in various scenarios. To this end, we executed several simulations that implement our cost model, introducing specific values to the model's parameters as depicted in Table I. Simulation results appear in Fig. 1.

As expected, we clearly observe that the number of user arrivals, namely λ, is one of the most important factors regarding the monetary costs of storage leeching. If we compare Fig. 1a with Fig. 1b and 1c, we discern that the value of λ is significantly different among them. Therefore, *the popularity of an application that exploits storage leeching plays a critical role on the economic costs that it will cause to providers*.

Clearly, the resulting costs of storage leeching are highly dependent on the type of application that is exploiting the system, which would consume distinct amounts of resources. However, *as the population grows, the consumption of download traffic seems to induce the major fraction of a provider's expense*, since it is proportionally more expensive than storage.

Regarding storage, we observe that a the value of f_s, which represents the fraction of users that contribute by creating storage accounts, considerably limits storage costs in case of being very low (Fig. 1a) or in case of a high storage consumption (Fig. 1c). In any case, note that users' data may be stored in the system for a long time, *which represents a long term and ever increasing monthly cost for the provider*.

We observe that, *even in case of modest acceptance of an abusive application, the economic costs may be important*. For instance, in Fig. 1a we observe that a small number of active users $(4,500)$ illicitly consume an amount of resources equivalent to $\$2,670$ after 90 days. In case of a large-scale abuse (Fig. 1b and 1c), these costs may reach dramatic numbers at short or medium term (e.g. $\$0.81$M).

We conclude that applications exploiting storage leeching are a potential threat to Personal Clouds, in terms of resource consumption and economic impact, even in case of modest user populations. Next, we prove our findings by implementing a file-sharing application that exploits storage accounts from multiple providers.

V. BOXLEECH: AN ABUSIVE FILE-SHARING APPLICATION

`Boxleech` is a proof-of-concept file-sharing application able to disseminate illegal or copyrighted content by abusing Personal Clouds. Essentially, it aggregates free accounts from multiple Personal Clouds into a single storage unit that can be freely accessed by users interested in a certain content. In particular, `Boxleech` aggregates free accounts from Dropbox, Box and SugarSync, three major storage vendors, which shows the potential impact of storage leeching and the simplicity to exploit public APIs to abuse Personal Clouds.

In Fig. 2, we provide a general overview of the functioning of `Boxleech`. We observe two Personal Clouds where a malicious user has registered a developer application and few free accounts. Besides, he enables the REST API access to these accounts obtaining the required tokens. Using the `Boxleech` client, he uploads chunks corresponding to an illicit content he wants to share. Finally, he generates and distributes the metadata file which contains the information to enable any other `Boxleech` user to download the content.

The design of `Boxleech` can be divided into three main blocks: *data management*, *metadata* and, *chunk assignment*.

Data management. First, similar to Dropbox and the likes, which internally do not use the concept of files, `Boxleech` splits every file into chunks of up to 100MB in size. There were three good reasons for this: i) To surpass the file size limitations commonly imposed in the REST API access to free accounts, ii) To exploit storage diversity by allocating chunks of the same file to different Personal Clouds and, iii) To make it impossible for a single provider to store an entire copy of an illicit content. Currently, `Boxleech` applies a simple fragmentation algorithm to create equally-sized file chunks. However, more elaborated mechanisms such as Erasure Codes [15] may be introduced to increase data availability via data redundancy.

Locally, `Boxleech` maintains an index which relates every chunk with the file it belongs to, as well as the information about the cloud account where it has been stored. To manage data chunks from these Personal Clouds, the implementation of `Boxleech` includes the client API of all of them. Clearly, supporting a new provider will require introducing the corresponding API implementation in the application.

Metadata. The objective of `Boxleech` metadata files (`.boxleech`) is to map a set of chunks corresponding to

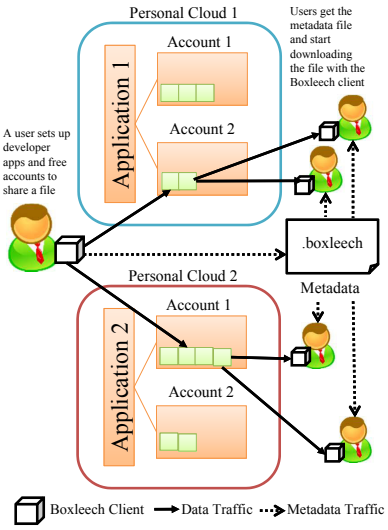


Fig. 2. Users abusing Personal Clouds by sharing illicit contents with Boxleech. Once users get the metadata file that contains the account access credentials for each chunk, they are able to download the shared content.

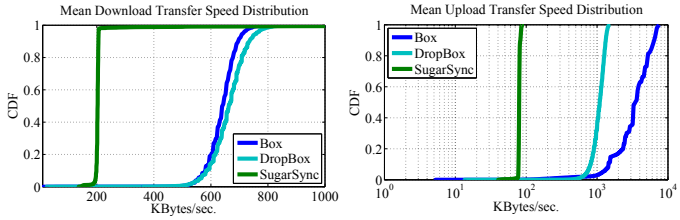
the same content to their location in diverse Personal Cloud accounts. A metadata file is formed by a set of rows, each one containing the following information for a data chunk: *[chunk_id], [order], [provider], [access_credentials]*. The first two fields describe the identifier of a chunk (e.g. hash value) and the chunk position, which is needed to reconstruct the file after downloading it. The *access_credentials* field includes the necessary access information to download that chunk from the appropriate provider. In the case of DropBox and Box, there is only need to include the access token for the account where the chunk is stored. However, SugarSync requires to include the secret/application keys as well as the account login/password information to renew the token after its expiration.

Boxleech is capable of generating a .boxleech file for a content that a user has uploaded, as well as to interpret these files to download contents shared by other users. Similarly to a *.torrent* file [16], there are several ways of indexing and distributing these metadata files, such as a Web server (tracker) or a Distributed Hash Table (DHT). Going further, we advocate for building a metadata index also exploiting Personal Cloud accounts. To illustrate this, let us assume that each metadata file is named with the *hash* of the original content name (e.g. film title). Making use of consistent hashing [17], we can partition the hash identifier space among a set of storage accounts, which adopt the role of traditional hash buckets. Hence, Boxleech clients are able to deterministically search for a hash value in the appropriate account. This leverages an integral file-sharing service entirely supported by exploited resources from Personal Clouds[2].

Chunk Assignment. The allocation of chunks when exploiting various Personal Clouds plays a critical role on the speed of transfers. To better explore this issue, we empirically measured the REST API service of DropBox, Box and SugarSync by actively uploading and downloading files from free accounts[3].

[2]In [9], Mulazzani et. al. point out that DropBox is being used to store and share *.torrent* files, as well as to distribute copyrighted material.

[3]Due to lack of space, the complete experimental methodology as well as the datasets can be found at http://ast-deim.urv.cat/trac/pc_measurement



(a) Download file MTS distributions. (b) Upload file MTS distributions.

Fig. 3. File mean transfer speed of Box, DropBox and SugarSync APIs.

In Fig. 3 we illustrate the empirical distributions of the obtained file *mean transfer speed* (MTS) values. This metric is defined as the ratio of the size of a file, S, to the time, T, that was spent to transfer it: $MTS = S/T$ (KBytes/sec).

Fig. 3 evidences an interesting fact: *Personal Clouds provide very disparate transfer speeds*. For instance, in Fig. 3b we observe that Box and DropBox provide a upload MTS several times faster than SugarSync —the same observation holds for downloads. Moreover, the heterogeneity of these services also depends on the *traffic type* (in/out). This can be appreciated comparing Fig. 3a and 3b: DropBox exhibits the best download MTS values whereas Box clearly provides the fastest uploads. Hence, we conclude that Personal Clouds are heterogeneous in terms of transfer performance.

Boxleech exploits this feature to show that leechers can obtain even faster transfers by intelligently allocating the file chunks to various providers[4]. This allocation depends on the *chunk assignment policy*. In Section VI, we propose and evaluate Boxleech using several chunk assignments.

Initialization. To share content with Boxleech, all we needed to do was to sign up for some free accounts and then register as a developer in each storage service. Once registered, we instantiated a fake application with the intention to receive an application and secret key pair. Using these keys, we validated our credentials and obtained the authorizing tokens that must be passed in every API call (see Section III). All this process was done with little human interaction since the core idea of the freemium model is to recruit as much users as possible through a simple sign-up process.

VI. Experimental Evaluation

Next, we evaluate Boxleech and we compare its performance with software clients delivered by DropBox and SugarSync to illustrate the potential benefits of storage leeching.

A. Setup & Methodology

Scenario. We executed our experiments in our university laboratories. We used 12 machines in order to run the different software configurations employed in our tests. We employed Intel Core i5 machines equipped with 4GB of DDR3 memory. The operating system was Windows 7 (DropBox, SugarSync clients) and Linux Debian (Boxleech). Machines were connected to the same switch via a Fast Ethernet link.

Software. *Personal Cloud Software Clients*. DropBox and SugarSync provide free and closed software clients to maintain

[4]We confirmed through experimentation that multiple parallel download transfers from a single data object do not decrease transfer performance. This provides an appropriate substrate to build an efficient file-sharing system.

in sync files from multiple devices and the Cloud[5]. In our experiments, both clients were explicitly configured to provide the maximum transfer capacity. Moreover, in the case of DropBox, we deactivated the LAN Sync option which permits the synchronization of multiple devices in the same network.

`Boxleech`. Our file-sharing application employed the standard API implementations to access storage accounts. Specifically, we used two configurations of free accounts in our tests: i) 3 free accounts, one for each Personal Cloud analyzed in this article and ii) 5 Box accounts, to test large storage operations to the same provider. `Boxleech` made use of parallel transfers when transferring chunks in and out from each account. In case of a failed storage operation on a chunk, it performed retries until making the operation succeed. This could increase transfer times in case of multiple failures.

Workload. For both software clients and `Boxleech` we executed an alternate upload and download workload. Basically, it consisted on generating a new file, uploading it to the account and downloading it before its deletion.

Specifically, in the case of software clients, the workload is executed by pairs of computers —each one dedicated either to upload or download files. First, the upload script created a synthetic file which is stored in the software client *watch directory* of the computer responsible for uploads. This script was continuously checking in the server-side whether the client finished the upload or not. In parallel, the download script was waiting in the second computer until the upload had finished. Then, it started measuring the download time until the remote file was available in that computer. When the download concluded, the download script deleted the file, which served as a notification to the upload script to repeat the process again.

Storage operations were performed over synthetic random and compressed files. This was necessary to prevent software clients from applying caching, deduplication and compression mechanisms over these files. Our results are based in approx. 100 storage operations for each software and configuration.

Chunk Assignments. To explore the impact of chunk assignments in depth, we performed a battery of Monte Carlo simulations over the *empirical data* collected in our experiment (see Fig. 3). Fig. 4 plots the impact of different chunks allocations. The abscissa axis shows the upload/download transfer time measured in seconds for transferring a file of $F = 600$ MB. For each possible allocation of n chunks among Box, DropBox and SugarSync, there is one corresponding bar in Fig. 4. Note that depending on the chunk size β, the number of chunks n will vary ($n = F/\beta$). The colored segments in the bars represent the time incurred to sequentially transfer the chunks assigned to a given Personal Cloud.

As expected, assigning more chunks to DropBox reduces the download time, since this vendor exhibited the fastest download capacity in our experiment. This always holds, irrespective of the chunk size. In any case, allocating the majority of chunks among Box and DropBox ensures to the abusive application good download performance while improving load balancing among both providers.

[5]Sugarsync client version: 1.9.71.94365.20120712. Dropbox client version 1.4.11. Box is excluded from this evaluation since it currently does not provide a free software client.

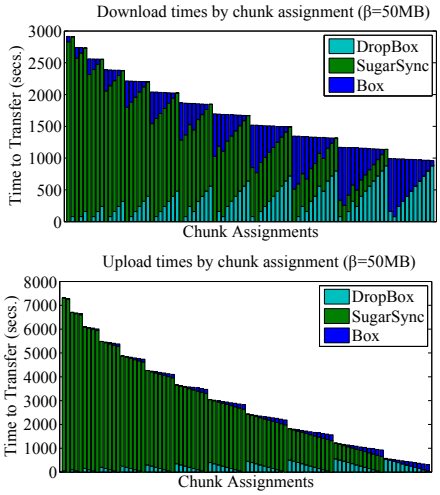


Fig. 4. Impact of chunk assignment on transfer times (chunks are assumed to be sequentially transferred). Clearly, different assignments report disparate transfer performance, which is essential to effectively exploit the service.

On the other hand, due to its poor performance, allocating more chunks to SugarSync yields higher upload times. The impact on transfer times of SugarSync uploads is much higher than in the case of downloads. In this sense, we observe an important improvement as more chunks are assigned to Box, which exhibits the fastest upload service in our experiment.

As a result of these observations, we implemented and tested three simple allocation policies to assess the potential benefits of exploiting storage diversity. These policies are:

- *Round Robin (RR)*: This strategy is extremely simple to implement and has been adopted in many real systems. This placement allocates the same amount of chunks to each user account in order to ensure fairness and reliability. This policy serves as a performance baseline and it does not make use of any source of information.
- *Upload/Download Proportional (UP, DP)*: Based in our analysis, we propose two new placements to reduce upload and download transfer times. Both placements assign a number of chunks in proportion to the transfer capacity of each Personal Cloud. The transfer capacities has been extracted from our measurement study, and therefore, UP and DP are informed assignment policies.

Next, we evaluate the differences in performance between both types of placement policies (informed and non-informed).

B. Experimental Results

Single provider. One simple form of storage leeching is to aggregate free accounts from the same provider. In the next experiment, we want to verify if the aggregation of accounts from the same provider entails some performance degradation. For this reason, we aggregated 5 Box accounts and uploaded large amounts of data. The results are shown in Fig. 5.

Fig. 5 shows the storage and retrieval performance of `Boxleech` aggregating 5 Box accounts for different amounts of data. Although the linear behavior of transfer times was expected, it is surprising to see the upload speed `Boxleech` with this configuration. Actually, the average transfer speed of

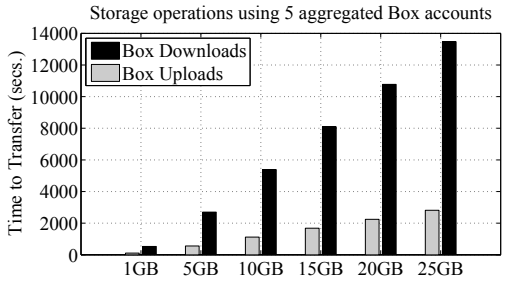


Fig. 5. Transfer performance of `Boxleech` aggregating 5 Box accounts. We observe that the upload capacity of Box is really high and can be effectively exploited to store and share large amounts of data.

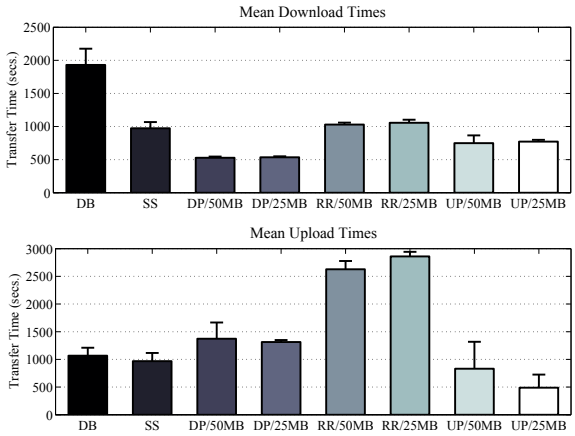


Fig. 6. Mean transfer times and standard deviation (error bar) of `Boxleech` under distinct configurations and DropBox (DB) and SugarSync (SS) clients.

chunks was $\approx$ 11.5MBps[6], which is a high-quality free service. One of the most important conclusions of this experiment is that *aggregating an arbitrary number of free accounts is extremely easy*. Furthermore, aggregating several accounts of the same Personal Cloud does not seem to degrade the service performance, meaning than exploiting a single provider is a feasible leeching strategy.

As an important remark, note that *a single user is able to consume around 25GB of storage and upload traffic, as well as 5GB of download traffic in one hour*. Thus, one can easily imagine the economic expense in terms of consumed resources that a large user population may cause to a provider.

Multiple providers. Next, we focus on the transfer speed of `Boxleech` compared with DropBox and SugarSync software clients. For this experiment we used 600MB synthetic files, which emulates a scenario of users sharing music albums.

In Fig. 6 we infer that `Boxleech` obtains better transfer speed than DropBox and SugarSync clients in many configurations. For downloads, `Boxleech` using the Download Proportional policy (DP) provides a transfer speed nearly *2 times higher than the obtained by the SugarSync client*, which is the client exhibiting fastest downloads. To wit, the DP policy assigns more chunks to Box and DropBox services, which present the highest download speeds in our measurement. This makes `Boxleech` downloads considerably faster. Note that

[6]Note that such a speed cannot be continuously maintained since we start a new TCP connection for each chunk to be transmitted.

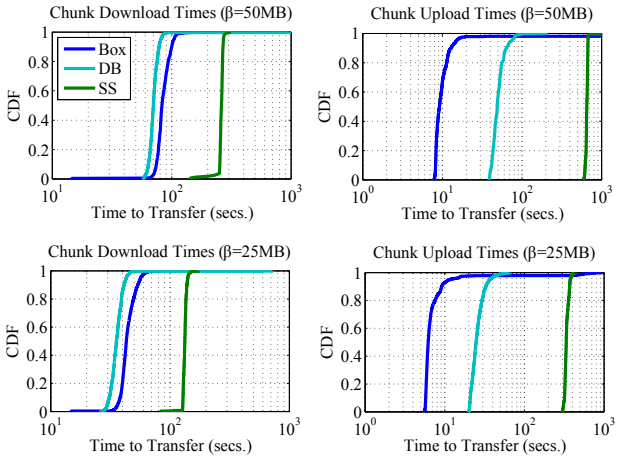


Fig. 7. `Boxleech` chunk transfer times distributions for both uploads and downloads, as well as for different chunk sizes β. Probably, due to the management of parallel transfers of `Boxleech`, a small fraction of chunks present really large transfer times.

even using the simple Round Robin (RR) policy, `Boxleech` reports a download speed similar to the SugarSync client.

For uploads, we see that `Boxleech` is able to obtain comparable or even better transfer speed than Personal Cloud software clients. That is, `Boxleech` using the Upload Proportional (UP) policy with chunks of 25MB presents upload times over 55% shorter than its counterparts. In this sense, the RR policy reports the worst performance due to the amount of chunks uploaded to the SugarSync service, which is really limited. However, this is the only policy that provides storage balance among accounts. Thus, there is a *trade-off between storage balance and transfer speed* when exploiting accounts from multiple providers.

Appreciably, for both uploads and downloads, we see that our informed chunk assignment policies provide a higher transfer speed than the RR policy. Hence, *the information of our measurement helps to better exploit storage diversity*.

Fig. 7 helps to understand the reported file transfer times in Fig. 6. Fig. 7 shows the chunk transfer time distributions for each Personal Cloud used by `Boxleech`. We found that a *small fraction of chunks* exhibit really large transfer times. This phenomenon is specially pronounced in Box uploads. This impacts on file transfer times of our application since all chunks should be transferred before finishing. This effect might be induced by the management of parallel transfers in `Boxleech`, that should be carefully addressed for those applications which want to optimize transfers.

Moreover, Fig. 6 shows that the chunk size (β) does not have important implications to the transfer performance, at least in case of moderate file sizes.

Another interesting observation comes from the analysis of DropBox and SugarSync clients (Fig. 6). In our experiments, DropBox exhibits a much greater REST API transfer speed than SugarSync. However, we see that the transfer performance of both clients is quite similar in case of uploads. Furthermore, we observe that SugarSync provides a download speed much better than the DropBox client. This suggests that: i) these clients may implement bandwidth control mechanisms in order to *restrict the resource consumption coming from*

free accounts, and ii) the performance of *REST APIs is not necessarily related with the performance of the software client*.

VII. Discussion

In this work, we just scratched the surface of the set of exploitation possibilities that the *storage leeching problem* permits. In addition to benefit abusive applications like `Boxleech`, storage leeching is a vector to materialize many other threats, such as denial-of-service attacks against Personal Clouds [4] or fraudulent resource consumption [7], [8].

In this sense, we highlight the danger that a quick account registration process (e.g. no *captchas*) represents to Personal Clouds. As a lesson learned from working with Personal Cloud APIs, we created over 140 free accounts and 35 developer applications of various vendors in few hours. This represents a virtual storage capacity of around 450GB. Moreover, it is not hard to imagine expert hackers creating scripts to facilitate, even more, the initial registration process for non-expert users. In our view, leveraging storage leeching to the masses would have important economic implications to Personal Clouds.

Although introducing countermeasures to provide a secure API service is strategic decision from a vendor's viewpoint, we propose the following ones based on our experience:

- *Enforce accountable user identities*: The main requirement to access free storage and register an application as developer is an email account. Thus, if email accounts are easy to create, any user can rapidly gather an arbitrary amount of free storage. We suggest to introduce filters in the creation of Personal Cloud free accounts and/or registering applications to enforce that one user obtains one account (phone number, human intervention). Currently, systems like Facebook introduce very restrictive procedures to their developer environments.
- *Expiration time for developer applications*: To discourage malicious users to exploit open APIs as a durable storage substrate, we believe that introducing expiration mechanisms to both developer applications and the related free accounts could be an effective countermeasure. By doing this, Personal Clouds would force abusers storing their data in the system to periodically migrate it.
- *Identify anomalous workloads*: Personal Clouds could benefit from research efforts focused on identifying fraudulent resource consumption to detect abuse in storage accounts related to developer applications. This suggestion comes from our empirical experience: we actively performed tests against free accounts for 2 months. In that time, we have not detected any change in the service provided by Personal Clouds, even though the executed workload could be easily detected as an anomaly.

Finally, it is surprising that many vendors do not implement this kind of security measures in their API service, even though it is technically possible. Perhaps, Personal Clouds assume the risk of a possible abuse of their service motivated by luring as many users and developers as possible. However, observing the behavior of SugarSync, where the REST API transfer performance is substantially worse than that exhibited by the software client, it seems probable that other providers will restrict the *freemium* API service in the future.

VIII. Conclusions

To lure customers and developers, Personal Clouds provide open REST APIs to create new applications that make their service even more attractive. However, the unintended consequence of this strategy is that it is very easy for a user to abuse the service by aggregating free accounts, from one or several providers, to obtain a high-quality storage service, what we term as the *storage leeching problem*.

We implemented a proof-of-concept file-sharing application to demonstrate how easy is to benefit from storage leeching. Our application is able to provide equal or better transfer QoS than Personal Cloud software clients, showing that free accounts can be easily exploited to obtain a practical Cloud storage service for free. We conclude that Personal Clouds should seriously consider the implicit risks of their free services and open APIs to avoid nefarious use from malicious parties.

Acknowledgment

This work has been partly funded by the Spanish Ministry of Science and Innovation through projects DELFIN (TIN-2010-20140-C03-03) and RealCloud (IPT-2011-1232-430000) and by the European Union through project FP7 CloudSpaces (FP7 − 317555).

References

[1] F. Research, "The personal cloud: Transforming personal computing, mobile, and web markets," 2011. [Online]. Available: http://www.forrester.com/rb/Research/personal_cloud_transforming_personal_computing\%2C_mobile\%2C_and/q/id/57403/t/2
[2] [Online]. Available: http://en.wikipedia.org/wiki/Dropbox_(service)
[3] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services," *Computer Science - Research and Development*, vol. 24, pp. 185–197, 2009.
[4] "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1–11, 2011.
[5] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All your clouds are belong to us: security analysis of cloud management interfaces," in *ACM CCSW'11*, 2011, pp. 3–14.
[6] L. Vaquero, L. Rodero-Merino, and D. Morán, "Locking the sky: a survey on iaas cloud security," *Computing*, vol. 91, pp. 93–118, 2011.
[7] J. Idziorek and M. Tannian, "Exploiting cloud utility models for profit and ruin," in *IEEE CLOUD'11*, july 2011, pp. 33–40.
[8] J. Idziorek, M. Tannian, and D. Jacobson, "Attribution of fraudulent resource consumption in the cloud," in *IEEE CLOUD'12*, 2012, pp. 99–106.
[9] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E.Weippl., "Dark clouds on the horizon: Using cloud storage as attack vector and online slack space," in *USENIX Security*, 2011, pp. 5–8.
[10] J. Srinivasan, W. Wei, X. Ma, and T. Yu, "Emfs: Email-based personal cloud storage," in *NAS'11*, 2011, pp. 248–257.
[11] A. Traeger, N. Joukov, J. Sipek, and E. Zadok, "Using free web storage for data backup," in *StorageSS'06*, 2006, pp. 73–78.
[12] H.-C. Chao, T.-J. Liu, K.-H. Chen, and C.-R. Dow, "A seamless and reliable distributed network file system utilizing webspace," in *WSE'08*, 2008, pp. 65–68.
[13] E. Hammer-Lahav, "The OAuth 1.0 Protocol," http://tools.ietf.org/html/rfc5849, 2010.
[14] J. R. Douceur, "The sybil attack," in *IPTPS'01*, 2002, pp. 251–260.
[15] R. Gracia-Tinedo, M. Sánchez-Artigas, A. Moreno-Martínez, and P. García-López, "FRIENDBOX: A Hybrid F2F Personal Storage Application," in *IEEE CLOUD'12*, 2012, pp. 131–138.
[16] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, 2003, pp. 68–72.
[17] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *ACM STOC'97*, 1997, pp. 654–663.