CloudSafetyNet: Detecting Data Leakage between Cloud Tenants

Christian Priebe
TU Braunschweig

Divya Muthukumaran
Imperial College London

Dan O'Keeffe
Imperial College London

David Eyers
University of Otago

Brian Shand
NCRS, Public Health England

Ruediger Kapitza
TU Braunschweig

Peter Pietzuch
Imperial College London

ABSTRACT

When tenants deploy applications under the control of third-party cloud providers, they must trust the provider's security mechanisms for inter-tenant isolation, resource sharing and access control. Despite a provider's best efforts, accidental data leakage may occur due to misconfigurations or bugs in the cloud platform. Especially in Platform-as-a-Service (PaaS) clouds, which rely on weaker forms of isolation, the potential for unnoticed data leakage is high. Prior work to raise tenants' trust in clouds relies on attestation, which limits the management flexibility of providers, or fine-grained data tracking, which has high overheads.

We describe *CloudSafetyNet (CSN)*, a lightweight monitoring framework that gives tenants visibility into the propagation of their application data in a cloud environment with low performance overhead. It exploits the incentive of tenants to co-operate with each other to detect accidental data leakage. CSN transparently adds opaque *security tags* to a subset of form fields in HTTP requests, using a client-side JavaScript library. Socket-level *monitors* maintain a log of observed tags flowing between application components. Tenants retrieve their logs and identify foreign tags that indicate data leakage. To check the correct operation of CSN, tenants send *probe requests* with known tags and verify that monitors are logging correctly. Using an implementation of CSN deployed on the OpenShift and AppScale PaaS platforms, we show that it can discover misconfigurations and bugs with a negligible performance impact.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls; C.2.4 [**Distributed Systems**]: Client/Server

Keywords

Cloud; Inter-tenant isolation; Data leakage detection; Socket-level monitoring

CCSW'14, November 7, 2014, Scottsdale, Arizona, USA.
ACM 978-1-4503-3239-2/14/11.
http://dx.doi.org/10.1145/2664168.2664174.

1. INTRODUCTION

Cloud computing enables organisations to deploy scalable web applications with little upfront investment but it removes control: organisations must trust cloud providers with their data and rely on the correct operation of security mechanisms for inter-tenant isolation, resource sharing and access control. Surveys show that organisations are more likely to adopt a cloud model if they are given assurances that their data remains secure [74, 48].

While cloud providers have an incentive to protect tenant data, they must correctly configure, deploy and manage a complex cloud software stack. The rise of reported occurrences of data leakage in cloud environments [15] shows that accidental misconfigurations due to human error or software bugs in cloud platforms are a real threat to tenant data. It is challenging for tenants, however, to find out what security mechanisms cloud providers have in place [91]. In particular, *Platform-as-a-Service (PaaS)* clouds, such as Google App Engine [36] or OpenShift [62], hide internal deployment details [18], giving tenants no visibility into the PaaS platform on which to anchor their trust.

From a security perspective, PaaS platforms are more vulnerable to data leakage because they typically rely on weaker forms of *inter-tenant isolation*, such as process containers [54, 66] or application-level virtualisation [19, 38]. They also have *shared components* used by multiple tenants such as load-balancers [11] and data stores [40]. While hypervisor-controlled virtualisation with *virtual machines (VMs)* provides hard isolation boundaries between tenants, PaaS platforms can leak tenant data through simple misconfigurations such as incorrectly set-up namespaces, isolation failure for shared components or incorrect request routing.

Due to the opaque nature of PaaS clouds, tenants cannot discover when data leakage has occurred and instead must depend on the security procedures of cloud providers. Cloud providers, however, are business-driven and may delay alerting affected tenants of a security incident because they want to protect their reputation or carry out a complete damage assessment first [55, 77].

Previous work has attempted to mitigate data leakage in clouds by offering stronger security guarantees [80, 81]. Such approaches, however, remove control from cloud providers over their infrastructure and thus struggle to gain widespread adoption. Techniques for data flow tracking [68] add independent mechanisms for observing the flow of data through a cloud environment and can mitigate data disclosure before it occurs. The drawback is that they incur high overheads when tracking data at a fine granularity [68] and typically require changes to the cloud platform and applications [60].

In contrast, we observe that, since failures of inter-tenant isolation potentially affect multiple tenants in a cloud environment, there is an incentive for tenants to *collaborate* for detecting data leakage. Collaborative approaches for threat detection have proven successful in various domains, including network security [49], finance [30] and retail [72], because they offer benefits to participating organisations, such as reduced individual costs.

Instead of preventing data leakage, our goal is therefore to provide a *monitoring system* that can discover data leakage between tenants in a PaaS cloud. The system should (a) not interfere with the management operations of cloud providers in order to be easily deployable with today's PaaS platforms; (b) not have a high performance impact in terms of the throughput and latency of processed web requests; (c) be robust against misconfigurations even if they are caused by the cloud provider; and (d) enable tenants to do damage control after a data leakage incident.

We describe **CloudSafetyNet (CSN)**, a collaborative monitoring system that examines the data flows of distributed tenant applications in a PaaS cloud in order to discover occurrences of data leakage. The idea behind CSN is that each tenant includes *security tags* in a subset of its client requests in a web application, which are then logged by a set of network *monitors*, distributed throughout the cloud environment. When tagged data breaches the isolation boundary of a tenant and is observed by another tenant's monitor, it indicates data leakage. The CSN approach makes few assumptions about the internals of a PaaS platform and is thus compatible with a wide range of existing platforms.

In more detail, CSN consists of three parts:

Client-side data tagging. To add security tags to data in web requests, CSN uses a JavaScript library, which is imported automatically as part of the client-side code of a deployed web application. The CSN library adds opaque security tags to *functionally taggable* text fields in an HTTP request, i.e. ones that are not affected by tags. A security tag consists of an application identifier and part of the field data, encrypted under the tenant's public key. A subset of all data is tagged to limit the performance impact of tag monitoring.

Socket-level tag monitoring. Tenants or the cloud provider deploy a set of *monitors* that, through dynamic library pre-loading, intercept TCP flows between processes of a distributed web application on the PaaS platform. Monitors are associated with application components, such as a tier in a multi-tier web application. Since monitors only observe incoming and outgoing network flows *passively*, they incur a low performance impact.

Each monitor records observed tags in a *tag log*, which is periodically retrieved by the tenant. To ensure the correct operation of monitors, tenants can issue *probe requests*. These requests contain a known set of *probe tags* that are indistinguishable from regular tags. A tenant can check for the existence of probe tags in their logs to verify the correct operation of the monitors.

Inter-tenant data leakage detection. Tenants attempt to decrypt the tags in their logs, and foreign tags that cannot be decrypted may belong to other tenants, indicating possible data leakage. Tenants cooperate with each other to confirm the source of leakage: they advertise foreign tags to other tenants out-of-band, e.g. through a mailing list, thus notifying the tenant whose data has leaked.

We deploy a prototype implementation of CSN on the OpenShift [62] and AppScale [14] PaaS platforms running on top of CloudStack [17]. Using real-world web applications (Fofou [28] and WordPress [92]), we show that CSN can detect data leakage caused by several representative classes of misconfigurations and software bugs. We also demonstrate that the CSN monitors incur only a low reduction in request throughput, even with a large number of tracked tags.

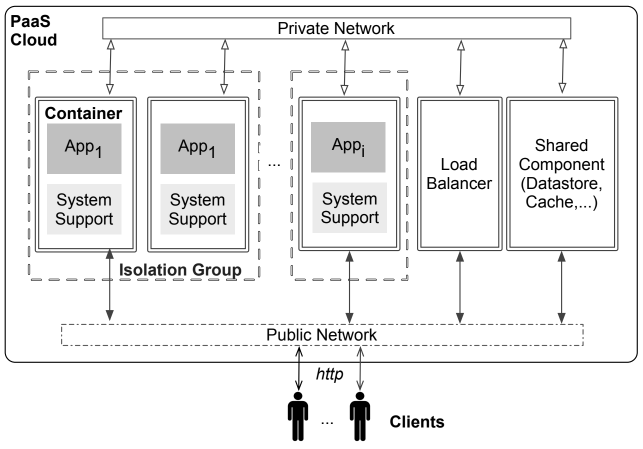

Figure 1: Overview of the architecture of a PaaS cloud, showing web applications deployed in containers

2. ACCIDENTAL DATA LEAKAGE

Over the years, a wide range of PaaS platforms have been developed, including Google App Engine [36], OpenShift [62] and CloudFoundry [16]. In a PaaS platform, As shown in Fig. 1, a PaaS platform can host applications belonging to multiple *tenants*. An application consists of distributed *application components*, such as front-end web servers, application servers and database back-ends.

Each application serves a set of *clients* that make requests and get responses. Data from clients is submitted through *web forms*, which are translated into *parameters* of HTTP `GET` or `POST` request methods to tenant applications. HTTP requests are sent to a web server residing in the PaaS cloud.

The components of applications are deployed on a set of virtualised nodes. Access to system resources such as file systems, memory and the network needs to be segregated correctly between different tenants. Typically this is done using a *container* mechanism [12], such as Linux containers [54]. All containers belonging to a given tenant form an *isolation group*—application data may flow between containers within the same group but should not flow from one group to another.

Tenants therefore want to discover any lapses in security that may lead to such data leakage between isolation groups. A flaw in the isolation mechanism can potentially affect any tenant—this provides an incentive for tenants to cooperate in order to detect data leakages.

2.1 Threat model

As business entities, cloud providers are conscious of their reputation among tenants. It is therefore in their interest to follow security best practices when managing their infrastructure to avoid data leakage between tenants. However, despite their best intentions, cloud providers are not immune to security incidents. Past press coverage [25, 69] shows that they are susceptible to occurrences of data leakage caused by (i) *accidental misconfiguration* due to human errors or (ii) *unknown software bugs* as part of the PaaS platform (see §2.2).

In the event of a data leak occurring, however, the interests of the cloud provider are at odds with those of the affected tenants—cloud providers may delay the notification of data leaks to their tenants either intentionally to protect their reputation or they may underestimate the importance of the incident altogether, not prioritising actions appropriately [74]. This prevents tenants from taking urgent remedial action, e.g. contacting financial institutions after credit card data was released.

To describe this threat model, we term cloud providers to be *"imperfect and selfish"* in dealing with security—i.e. susceptible to accidental data leaks and selfish about revealing such leaks to tenants. Note that we do not assume that cloud providers actively attempt to disclose tenant data. Since they have full control over the PaaS platform, tenants could only protect their data by encrypting it [75], but this would require changes to existing web applications, reduce performance and introduce complex key management issues.

In addition, we do not target the case of an active adversary who tries to exploit bugs in the cloud platform or the applications in order to steal data. This is an orthogonal problem, and these types of attacks can be dealt with by using existing boundary security approaches, such as firewalls.

Our approach can defend against malicious tenants or clients who introduce false positives into the leakage detection approach to slander the cloud provider (see §3.4). We cannot deal with the case in which multiple tenants collude to create artificial data leaks. Instead, by using a third-party service to authenticate tenants (see §3.3), we make such attacks harder to carry out, e.g. by preventing Sybil attacks.

2.2 Misconfigurations and bugs

Cloud providers manage the servers, network and storage that are necessary to deploy a PaaS platform. Next we describe representative, previously reported, classes of accidental misconfigurations or software bugs in PaaS platforms. Such issues may result in tenant data from one application leaking to another application, to the PaaS platform or the public Internet.

Isolation failure between containers. Containers offer weaker isolation than traditional OS-level VMs but also incur lower overhead by sharing portions of the host kernel and OS instance. Linux containers [54] and its derivative Docker [22] are used by a new crop of PaaS platforms [20, 23, 64], while other platforms [65, 16, 38] use custom container technology.

To support containers, globally-visible resources, such as process IDs, file systems and network interfaces, must be wrapped in a namespace layer, which provides each container with its own isolated view. Some aspects of Linux kernel namespace isolation are still evolving [51]—this lack of maturity carries an increased risk of undiscovered bugs in the software [26].

Accidental misconfiguration of container mechanisms makes it easy to violate the isolation boundaries between tenants. For example, OpenShift uses the `pam_namespace` mechanism [59] to provide poly-instantiation (i.e. multiple instances) of the `/tmp` and `/var/tmp` directories. Different applications on the same node can therefore use these directories as they normally would without interfering with each other.

The `pam_namespace` mechanism relies on a configuration file in `/etc/security/namespace.d/` to determine which directories need to be poly-instantiated. For example, the following configuration creates separate application-specific instances of the `tmp` directories under `$HOME/.tmp/`: (lines are wrapped for display)

```
/tmp $HOME/.tmp/ user:iscript=/usr/sbin/oo-namespace-init
      root,adm,apache
/var/tmp $HOME/.tmp/ user:iscript=/usr/sbin/oo-namespace-
     init root,adm,apache
```

If two applications created identical files under `/tmp`, they would thus not observe each other's files. A configuration mistake, such as omitting the above lines, however, would result in shared visibility of the files. If there were two applications that both accessed `/tmp/log.txt` without poly-instantiation, they would read each other's data, leading to an inter-tenant data leak.

Some PaaS platforms, such as Google App Engine and AppScale, achieve container isolation by applying white-lists to Java or Python classes or methods, thus preventing access to particular system calls [39]. Mistakes in the white-listing have resulted in inter-tenant data leaks, e.g. through access to the file system [10].

Isolation failure in shared components. To save resources, PaaS platforms, such as AppScale and Heroku, share application components, including load-balancers, caches and data stores, across tenants [38]. Shared components cannot rely on container isolation and instead must implement their own isolation mechanisms, e.g. based on namespace identifiers. For example, AppScale initially lacked namespace isolation for its *memcached* service, with the expectation that users want to share data between applications. It was pointed out that this could lead to inter-tenant data leakage [9].

Incorrect web request routing. A major benefit of PaaS platforms is that they allow applications to be scaled automatically or manually [63] by (de-)allocating resources based on the workload. In response to workload surges, the PaaS platform deploys the application on additional containers and uses a load-balancer to route requests between them.

Subtle bugs in the request routing logic may lead to unintended data leakage. For example, Amazon's Elastic Load Balancer [5], which routes requests dynamically between EC2 server instances, returns an IP address from a global pool of addresses. When an application is scaled down, the IP addresses are returned to the pool to be used by other applications. Due to DNS resolution caching, clients that ignore the *time-to-live* (TTL) set by Amazon continue using the old IP address, and their requests may be routed to the application of another tenant.

Incomplete data deletion. The file system accessible to applications can breach tenant isolation on PaaS platforms. When data is deleted by an application, it must be eliminated completely from all disks. Recently an incident of data leakage was reported for a cloud provider, which did not scrub block devices used by VMs. This permitted application components of future tenants to read the data of prior tenants [21].

Incorrect access control configuration. Access control mechanisms, such as SELinux [71] and AppArmor [8], are used by PaaS platforms to enforce access policies for containers when using files and processes. While permitting expressive policies, these policies can be complex to configure. A mistake in a policy configuration file may result in the incorrect labeling of resources, thus allowing data leaks. Cloud providers may also turn them off when they interfere with legitimate operations of tenant applications [47, 84].

All of the above cases involve some aspect of a breakdown of isolation between containers: data leaves the confines of one application's container and enters the container of another application. To detect such leaks, cross-container data propagation must be tracked to determine whether or not it constitutes a data leak.

2.3 Requirements and existing solutions

We outline the requirements of a solution that detects accidental data disclosure in PaaS clouds and describe why existing approaches do not satisfy all of these requirements. We discuss related work in more detail in §6.

R1: Ease of deployment. Any solution that detects data leakage should be easily deployable in today's PaaS environments. Existing *information flow tracking* (IFT) systems [83] can track the propagation of data through a system but require substantial changes to the OS or cloud stack, or make specific assumptions about the architecture of applications [52, 98, 58]. Approaches for remotely verifying the software installed on cloud platforms based on *trusted hardware* [80] require costly hardware and software upgrades in a cloud environment.

Another challenge is that existing PaaS platforms differ in the degree of control that tenants can exert over the underlying infrastructure. For example, some platforms permit tenants to execute arbitrary processes whereas others restrict actions through predefined APIs. This makes it difficult to determine how, where and by whom a data leakage solution can be deployed.

Our solution for data leakage detection should be compatible with existing PaaS platforms and web applications, and require minimal per-application modifications.

R2: Low performance impact. To convince cloud providers and tenants to adopt a given solution, it should not reduce substantially the performance of applications. Existing solutions typically try to achieve complete isolation between tenants and thus pay a high performance cost. For example, *fine-grained IFT* [68, 83] can reduce execution performance by orders of magnitude. *Accountable VMs* [42] can offer tenants assurance that their VMs execute correctly but require tenants to have local resources in order to reproduce all execution. Approaches that *encrypt* all tenant data can give strong security guarantees but incur costs of encryption and computation on encrypted data [35, 75].

We argue that it is acceptable to offer weaker security guarantees, which in turn have a negligible performance impact. Our solution should not attempt to prevent all data leakage but instead discover the majority of instances in a timely fashion.

R3: Robustness against misconfiguration. Since our threat model assumes that the cloud provider may introduce misconfigurations, we cannot assume that a given solution for data leakage detection is deployed and configured correctly. While approaches based on trusted hardware [80] can operate in untrusted environments, they withdraw control from the provider over their own infrastructure.

We want to provide a solution in which tenants can check the correct operation of data leakage detection without relying on external hardware support. In addition, data leakage detection itself should not jeopardise the confidentiality of data.

R4: Enable data leakage response. A tenant whose data has experienced data leakage should have a chance to respond to the incident before it becomes public, as in the case of vulnerability disclosure. Tenants may want to take quick remedial action, such as asking their clients to change passwords in a timely fashion.

Our solution should therefore not expose the identity of tenants who were affected by data leakage to other unaffected tenants.

3. CloudSafetyNet DESIGN

To meet the aforementioned requirements, we describe the design of *CloudSafetyNet (CSN)*, a system that monitors the data flow within a distributed tenant application on a PaaS platform to discover occurrences of accidental data leakage. The main idea behind CSN is to add *monitors* to applications in order to capture the propagation of a *subset* of all client data in network flows, which is *tagged*. If the subset is sufficiently large, tagged data is likely to be affected in the event of data leakage.

The monitors may be deployed either by cloud providers or tenants depending on the specific PaaS platform. The cloud provider would deploy monitors in platforms, such as AppScale, in which tenants have no control over the underlying infrastructure beyond the ability to upload their application code and set configuration options. Other PaaS platforms, such as OpenShift, are closer to infrastructure-as-service (IaaS) offerings—tenants exert greater control over the infrastructure. In such cases, they can deploy monitors independently, without cooperation from the cloud provider.

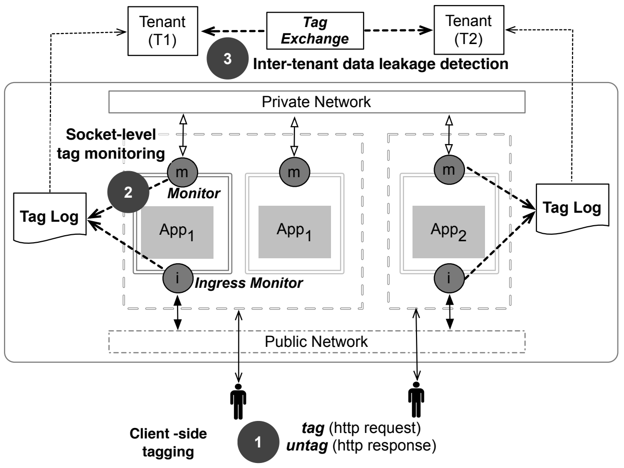

Figure 2: Design of the CSN monitoring system

Tenants then *cooperate* with each other to discover data leakage. Based on collaborative efforts in other fields, such as security threat information sharing [31, 89, 72], it has been shown that even competitive tenants find it mutually beneficial to share information that helps identify data leakage in a cloud platform. We use a trusted third-party service that authenticates tenants and allows them to exchange information about data leaks, similarly to software vulnerability databases [1].

As shown in Fig. 2, a deployment of the CSN system in a PaaS cloud consists of three main parts, which we introduce below:

(1) Client-side data tagging. Form-field data in HTTP requests is appended automatically with opaque *security tags* at the client-side. This is done using a JavaScript library that is imported as part of the client-side code of a deployed web application. Tags indicate to a monitor what to log, and they are encrypted with a tenant's public key. To avoid having to modify web applications in order to support in-band tags, CSN adds tags to a subset of form fields, which are *functionally taggable*, as explained in §3.1. By tagging only a subset of all data, the overhead of data flow tracking is reduced (requirement R2).

(2) Socket-level tag monitoring. Socket-level *monitors* intercept network flows between processes of a distributed web application on the PaaS platform. Monitors log the tags that they observe in a *tag log*. If foreign tags belonging to other tenants are recorded in a tag log, it may indicate inter-tenant data leakage.

Tenants can periodically issue *probe requests*, which contain a known set of *probe tags*. Since these probe tags should be present in the tag logs, they allow tenants to check if monitors operate correctly. This makes CSN robust against accidental misconfiguration by the cloud provider (requirement R3).

(3) Inter-tenant data leak detection. To detect foreign tags in their tag logs, tenants try to decrypt the tags. If the decryption fails, it indicates that the tag belongs to another tenant and must have breached the isolation group to have been observed by a monitor. Tenants exchange such foreign tags with each other out-of-band in order to confirm the source of data leaks cooperatively.

Fig. 3 illustrates the overall process of tagging, monitoring and leakage detection in a sample scenario with two tenants, tenant A and tenant B. The individual steps are discussed in detail in the following sections.

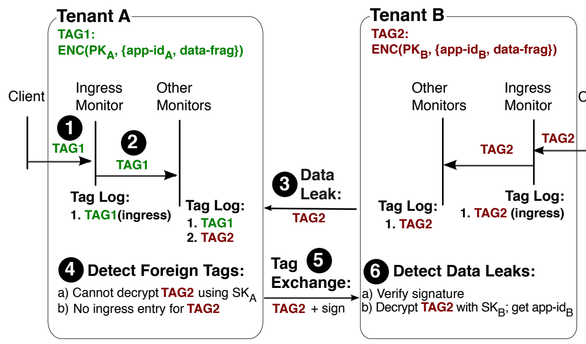

Figure 3: Tagging, logging and data leakage detection in CSN

3.1 Client-side tagging

Clients tag form-field data in web requests, which enables monitors to observe how this data propagates within the PaaS platform (see Fig. 2). The tagging approach is driven by the requirement that we want to make few if any changes to distributed web applications (requirement R1). As a result, CSN is limited to tracking form-field data that we call *functionally taggable*, which is defined by two conditions on how the application uses the data:

DEFINITION 1 (FUNCTIONALLY TAGGABLE). *Data $\mathscr{D}$ from a form field is considered* functionally taggable *in application $\mathscr{A}$ if (1) $\mathscr{D}$ is not modified by $\mathscr{A}$ and (2) appending data to $\mathscr{D}$ does not change the outcome of any operations performed on $\mathscr{D}$ by $\mathscr{A}$.*

Tags are constructed to be unique, i.e. they have low probability of occurring naturally as part of client requests. This means that applications can perform operations such as keyword searches on tagged data. For example, WordPress lets users create blogs through a web form with fields for "topic" and "content". The only operations performed on data in both fields is keyword-based search, making both fields candidates for tagging. On the other hand, fields that store dates or postcodes are not functionally taggable because they may be subject to comparison operations.

Prior work [76] has shown that a majority of functionality in today's web applications is related to storing and sharing of data. Therefore a significant amount of data processed by these applications, especially personal information, such as first and last names, addresses and posts, are functionally taggable.

Format of tags. Tags serve two purposes. They (i) indicate to monitors what to look for in intercepted data flows; and (ii) help tenants identify the source of a leak. A tag has the following format:

$$(\textit{tag-marker},\ \textit{ENC}(\textit{PK}_{\textit{tenant}}, \{\textit{app-id},\ \textit{data-fragment}\})$$

To keep monitors simple and efficient (requirement R2), we use a single *tag-marker* that all monitors search for. The marker is a unique string that has a low probability of occurring naturally as part of the application data.

To pinpoint the source of a data leak (requirement R4), tags relate back to the application and field data that they are associated with. They contain an *application identifier* (*app-id*) to determine which application the data belongs to, and a *data-fragment* of the original form-field data. This allows tenants to understand what data leaked and take remedial action. Since form field data can be large, only a small fixed portion of it is stored in the *data-fragment*.

To protect the information in the tag, each tenant has a public/private key pair (PK_{tenant}, SK_{tenant}), and the *app-id* and *data-fragment* are encrypted with the public key of the tenant PK_{tenant}. Encrypting the *app-id* serves two purposes: (i) it allows a tenant to determine if it can successfully decrypt the tag; and (ii) it hides the identity of the affected tenant in the event of a data leak (requirement R4). Encrypting the *data-fragment* protects against accidental data disclosure by the CSN system itself (requirement R3), e.g. due to exposed tag log files, and hides the data from other tenants during tag exchange (see §3.3).

Selection of tagged fields. The number of form fields that are tagged influences the probability with which CSN can detect data leaks—leaks of untagged fields go undetected. On the other hand, tagging every field impacts performance and the size of tag logs. The fact that we consider only functionally taggable form fields limits the number of fields that are tagged.

We assume that tenants have sufficient knowledge about their applications to determine which fields are functionally taggable. They specify all such fields to be tagged, which maximises the probability of data leak detection. Previous work [76] has shown that such fields can be identified automatically using program analysis.

Fig. 3 shows two tags, Tag 1 and Tag 2, that are used to tag form-field data of two applications belonging to tenant A and tenant B, respectively. Tagging is done by the CSN JavaScript library in a client's web browser (see §4).

3.2 Tag monitoring

Monitors execute within tenant containers and observe the network flows of the application. As shown in Fig. 2, each application component, such as a web front-end or a data store, has a monitor attached to it that tracks incoming and outgoing data. Monitors examine intercepted data to search for *tag-markers*, as introduced by the client-side. If a monitor observes a marker, it writes the encrypted tag data that follows it to a tag log.

To detect a data leak, a tenant must determine if data entered its application through valid means, i.e. through one of its designed entry points such as a web server that accepts client requests. We assume that leaked data does not enter through such entry points. Monitors that are deployed at the entry points to an application are termed *ingress monitors*, e.g. they observe incoming client HTTP requests from the public network (see Fig. 2). Depending on the application deployment, there may be multiple ingress monitors.

Fig. 3 shows the order in which tags that are introduced at the client-side are observed, first by the ingress monitor (step 1) and then by other monitors (step 2).

Socket-level monitoring. Our objective is to monitor all incoming and outgoing tenant data in an application without making changes to the application and without being tied to the specifics of a given PaaS platform (requirement R1). We assume that data that leaks to another tenant eventually is observed at the network level, either as a response to a client request, or when it is processed by an application component residing in a different container.

To detect inter-tenant data leaks, we want to know if a tag was observed entering or leaving a particular isolation group via the network. We therefore deploy monitors using *socket-level interception* within each tenant container. Sockets sit at the entry and exit points at which network data enters and leaves a container.

Tag logs. As illustrated in Fig. 3, monitors record the tags that they observe in a *tag log*. It stores the encrypted portion of the tags without further processing. This design enables monitors to be kept deliberately simple—since they passively observe network flows, they do not introduce a substantial overhead during application execution (requirement R2).

Below we give an example of the entries in a tag log. It contains the encrypted *app-id* and *data-fragment*. In addition, if a log entry was generated by an ingress monitor, an *ingress* flag is set to 1 to identify it as part of a client request from the public network:

```
MWRkNmUyY2VhZGQ5NWFlODUwNGI0M2UyYmFiNDEyY... 1
NKosty42GhITQ3NW2BTODUwJGL8MKUpYrFiIEyYPP... 0
MWRkNmUyY2VhZGQ5NWFlODUwNGI0M2UyYmFiNDEyY... 0
```

Probe requests. Monitors themselves may be misconfigured or deployed incorrectly, e.g. a cloud provider may accidentally deactivate a monitor. Tenants should therefore be able to check if their monitors operate as expected (requirement R3). They cannot assess the correct operation of monitors by considering the tag logs alone because they do not know the requests issued by external clients.

As a solution, tenants periodically have a special client issue *probe requests*, which are valid web application requests but contain a set of *probe tags* only known to the tenant. Probe tags lead to a deterministic sequence of entries in the tag logs of an application. After issuing a probe request, a tenant can check if the logs contain the corresponding probe tags. If probe tags are missing, tenants know that the monitors are not functioning correctly, and they can take remedial action. From the perspective of monitors, however, the probe tags are indistinguishable from regular tags.

The frequency of probe requests is decided by the tenant and depends on how much they trust the correct deployment of the CSN system, and how much of their application's resources they can afford to allocate towards processing of probe requests.

3.3 Data leakage detection

Tenants periodically retrieve the tag logs of all monitors that are part of their isolation group. They audit the logs by trying to decrypt each log entry using their private tenant key SK_{tenant}. A successful decryption indicates that the tenant can see their own *app-id*. If decryption fails, this tag could have leaked from another tenant's application. The tenant then has to validate that decryption failure is due to a genuine data leak, as described below.

Validating data leaks. Since a tenant's public key PK_{tenant} and *app-id* are known publicly, a malicious client could spoof data leakage to damage the reputation of a cloud provider. For example, a malicious client of tenant A can send a request with a tag that contains the *app-id* of tenant B and is encrypted using B's public key PK_B. When tenant A is unable to decrypt the tag, it would assume that this is a leaked tag.

To protect against this, a tenant checks if the tag was observed by an ingress monitor. If there is a corresponding entry for the tag with the *ingress* flag set, the tag originated from one of the clients of that application and therefore does not constitute a data leak.

In Fig. 3, we see that, due to a data leak (step 3) from tenant B to tenant A, Tag 2 is present in tenant A's tag log. Tenant A determines that Tag 2 is a foreign tag because the tenant is unable to decrypt it and does not have an ingress entry for it (step 4).

Tag exchange. After a tenant discovers a foreign tag without an ingress entry, it shares the tag out-of-band with other tenants, who can determine if the tag belongs to their application. We propose using a trusted third-party service that authenticates tenants and allow them to post data leaks in a database, which can be searched by other tenants, similarly to vulnerability databases for software bugs [1, 3, 2]. Authenticating tenant identities makes them reputation-conscious and deters false claims. It also prevents a malicious tenant from creating multiple identities in order to post and confirm false data leaks, slandering the cloud provider.

In Fig. 3, tenant A shares Tag 2 through the tag exchange mechanism (step 5), and it is received by tenant B. Tenant B checks the signature, decrypts the tag with its secret key SK_B and finds its own *app-id* (step 6). This indicates that it was the source of the data leak.

3.4 Discussion

Does the tag log grow too large? Over time, the sizes of tag logs will grow. Tenants can control log sizes by selecting the number of tagged form fields. However, the actual workload can affect log sizes beyond tenant control. For example, a `SELECT` query in WordPress may return 100 responses based on a keyword match of post titles. If CSN tags the "title" form field, this results in 100 corresponding entries in tag logs.

There are ways in which log sizes can be reduced: (i) duplicate entries need not be stored in tag logs, because tenants need only observe the first occurrence of a tag; and (ii) tag logs may be rotated after tenants have audited them: once a tenant has checked a tag, there is no further need to store it (unless the cloud provider requires the log for dispute resolution).

Ingress tag entries from logs, however, must be stored for longer because they help determine if a data leak is genuine. Typically the number of ingress tag entries is equal to the number of unique client requests made to an application. Since only existence testing is required for ingress tag entries, they may be compacted using Bloom filters [13].

Does CSN detect all data leaks? CSN detects leaks only along monitored data paths. We made a deliberate design choice to place monitors as part of the application components at the network level, because, ultimately, the data must be seen at the network before it reaches clients. For example, a file-system leak between two colocated tenants will be detected by our approach if and when that data reaches the network.

To gain greater visibility into data propagation in a cloud environment, monitors may also be deployed in other parts of a PaaS platform such as file system implementations, shared data stores, caches, etc. This would require, however, more substantial changes to the PaaS platform.

Even when a monitor observes network traffic between application components, if that traffic is encrypted, the monitor is unable to detect tagged data. In particular, with HTTPS client connections, socket-level ingress monitors will not observe tags. Encrypted traffic can be managed by intercepting communication at higher layers of the software stack. For example, to handle HTTPS ingress communication, a library pre-loading mechanism could intercept at the TLS/SSL layer in the webserver instead of the socket level [29], or traffic could be monitored by a reverse proxy.

Finally, CSN can only handle cases where tags propagate unmodified with the data, including when data is exposed through a data leak. We justified this assumption in §3.1.

Can an attacker spoof data leaks? An attacker may try to slander the reputation of a cloud provider by spoofing a data leak using CSN. Since all the information needed to construct a tag is public, an attacker may act as a client of two tenant applications and try to introduce the same data with the same tag into both applications. As described in §3.3, we can detect such an attack using ingress logs: when a tenant cannot decrypt a foreign tag, it validates it as a genuine data leak by ensuring that there is no corresponding ingress log entry.

If an attacker can gain control of a tenant, it could try to act as a client of another tenant's application, insert a tag into that application and introduce the same tag into its own application out-of-band. To prevent this attack, ingress monitors have to be extended to add a random nonce to each observed tag. This permits tenants to ignore foreign tags received through the tag exchange mechanism

```
1   <html>
2   <head>
3   <script_src="csnlib.js"></script>
4    ...
5   </head>
6   <body>
7   <form action="some.php" class="csn-tagged-form">
8      <input type="text" name="title">
9      <input type="text" name="full_name"
            class="csn-tagged-field">
10     <textarea name="comment"
            class="csn-tagged-field"></textarea>
11     <input type ="submit" value="Submit">
12   </form>
13   ...
14  </body>
15  </html>
```

Figure 4: Use of the CSN JavaScript library for client-side tagging

that decrypt successfully but whose nonce does not match an entry in their ingress logs. Preventing this attack comes at the cost of making monitors active, thus increasing their performance impact.

The above strategy cannot defend against an attacker that is permitted to gain control of multiple, colluding tenants. However, such an attacker can only demonstrate spoofed data leaks between applications that they control and not applications of other tenants.

How many participating tenants does CSN need? The success of CSN in detecting a given data leak depends on how pervasive the leak is and the number of tenants participating in CSN that will report a leak. The probability p that a data leak is reported is:

$$p = 1 - \binom{n-j}{k} \Big/ \binom{n}{k}$$

where n is the total number of tenants in the cloud environment, j is the number of participating CSN tenants and, k is the number of tenants affected by the leak. For example, if 15% of tenants participate in CSN, and a given data leak affects a random 15% of the tenants, the probability of the leak being reported is 93%. Therefore, even with a small incremental deployment of CSN, tenants benefit from better security.

4. IMPLEMENTATION

Our implementation of CSN consists of three main components: (i) a JavaScript library for client-side tagging; (ii) a library that implements socket-level monitoring; and (iii) a service for clients to retrieve tag logs.

Client-side tagging library. To support a wide range of modern web applications, we use JavaScript as a basis for implementing client-side tagging of data. Our client-side tagging library has only 160 LOCs and uses the `pidCrypt` library [73] for encryption support needed to generate security tags. We use RSA for encryption with a 2048-bit key and apply salt to strengthen the encryption's protection against cypher-text matching attacks. We assume the use of an encryption scheme that maintains key-privacy.

Ahead of deployment of a given web application, a tenant needs to decide on the set of fields in a web form to extend with security tags (see §3.1). Web pages that include sensitive form fields have to be modified as described in the following before they are deployed as part of the tenant's application.

The listing in Fig. 4 shows an example of an HTML page, with the modifications to add security tags using our JavaScript library underlined. First, the tagging library is added to a web page (line 3). The `class` attribute is used to indicate that a form is subject to data leakage detection (line 7). Form fields that should include a tag are marked with another `class` annotation (lines 9–10).

The creation of security tags occurs after the form was filled with data in a client's web browser and is about to be submitted to the application's web server. Our tagging library intercepts `onSubmit` events of forms that are annotated with the `csn-tagged-form` class and adds security tags to annotated form fields. For this, the CSN library has to include the tenant's public key PK_{tenant} and the *app-id* required for generating application-relatable security tags.

Similarly, when data is returned from the web server as part of an HTTP response, security tags must be removed from the data. The tagging library registers a handler for `DOMContentLoad` events, which are issued by the browser when a page is loaded. Upon receiving this event, the library traverses the Document Object Model (DOM) [24] tree and removes tags from content elements before the page is displayed.

Besides handling the transparent addition and removal of tags for web forms, the tagging library also exposes two functions, `tag()` and `untag()`, which developers can use to control the tagging of non-form data explicitly, e.g. when using AJAX [34].

Socket-level monitor library. We want to provide an implementation of monitors that can observe all incoming and outgoing data to and from a tenant container (a) independently of the specifics of a given PaaS platform and (b) with a low performance impact. We implement monitors as a C-library, which intercepts socket connections at a process level and monitors network streams for security tags. This makes it easy to support existing applications without requiring kernel-level changes.

The monitor library intercepts data of a process by wrapping the `libc` library functions responsible for socket communication (i.e. `read`, `write`, `send` and `recv`). The library is pre-loaded at process-creation time using the `LD_PRELOAD` mechanism. This makes it straight-forward to integrate it with processes used by existing PaaS platforms such as OpenShift. OpenShift starts components such as MySQL through start-up scripts, which are modified to load monitors by setting two environment variables, `LD_PRELOAD` to preload the CSN library and `CSN_LOG_FILE` to specify the location of the tag log file to be used by the monitor:

```
LD_PRELOAD=/usr/lib/libcsnmon.so \
CSN_LOG_FILE=$OPENSHIFT_DATA_DIR/csn/mysql.log \
usr/bin/mysqld_safe --defaults-file= \
 $OPENSHIFT_MYSQL_DIR/conf/my.cnf > /dev/null 2>&1 &
```

5. EVALUATION

Our evaluation goals are to investigate the ability of CSN to detect instances of data leakage and to measure its impact on performance. We reproduce the different classes of data leakage, as discussed in §2.2, and investigate how CSN is able to detect them. We also explore the performance of CSN under a realistic web application workload and in micro-benchmarks.

5.1 Experimental set-up

All experiments use a set of 4 GHz VMs with 6 GiB of RAM running as part of a local CloudStack deployment. The operating system is either CentOS 6.4 64-bit for experiments that relate to OpenShift or Ubuntu 12.04 64-bit for all other experiments. For OpenShift, we use version 2 of the open-source implementation OpenShift Origin; experiments with AppScale run on version 1.11.

5.2 Data leakage detection

Table 1 summarises the instances of misconfigurations and bugs that we evaluate that cause data leaks between tenants' applica-

Table 1: Instances of misconfigurations and bugs that can lead to accidental inter-tenant data leakage. The rightmost three columns indicate the particular platform used in experiments.

Class	Instance	Affected platforms	Platform used	Description	Data leak
Isolation failure between containers	Shared kernel namespace	OpenShift, CloudFoundry, Heroku, Dokku, Deis, Flynn	OpenShift	Omitted `/tmp` config line in `pam_namespace`	Apps read same file in `/tmp`
	Language/library whitelisting	Appscale, App Engine			
Isolation failure in shared component	Shared cache namespace	App Engine, AppScale, Windows Azure	AppScale	Removed app id from `_GetKey` in `memcached`	App reads `/tmp` file of another app
	Shared data store namespace	Appscale, App Engine, Heroku, Force.com			
Incorrect request routing	Load balancer	OpenShift, AppScale, CloudFoundry	OpenShift	Wrong IP address in `haproxy.conf`	Requests forwarded to wrong app
Incomplete data deletion	Tenant data not deleted	All	OpenShift	Filesystem remounted with old data	App sees `/tmp` files of another tenant

tions. For each instance, we send a client request to the application in order to trigger the data leak and check if CSN can detect it.

Isolation failure between containers. We show that CSN detects data leaks caused by misconfiguring `pam_namespace` [59], which OpenShift uses to provide container-specific instances of the `/tmp` and `/var/tmp` directories. Without isolated instances of the `/tmp` directory, two deployed applications can end up using the same file, resulting in a data leak. To recreate this problem, we use a minimal PHP application, which accepts data from a single form field and writes it to a file named `/tmp/request.txt`. It then returns the contents of the file to the client.

Misconfiguration. To introduce this data leak, we omit the directive within `/etc/security/namespace.d/tmp.conf` that instructs `pam_namespace` to create isolated instances of the `/tmp` directory. We also disable SELinux on the node running the containers.

Tagging and deployment. We mark the form field of the application to be tagged at the client-side. We deploy two instances of the application in our OpenShift installation. Instance A receives a client request, then instance B receives one. Since there is only one container per application, each tenant has one tag log.

Leak detection. Instance A logs two tags: an ingress and a non-ingress tag for the request and response, respectively. It is able to decrypt both tags successfully.

Instance B, however, has three tags in its tag log: one ingress and non-ingress tag from its own client and a third tag, corresponding to instance A's data that leaked through the `/tmp` file. It was logged by the monitor when sent as part of the response to the client of instance B. Instance B is unable to decrypt this tag. Since there is no ingress entry for this tag, instance B advertises it as a data leak. Instance A decrypts the tag and confirms the data leak.

Isolation failure in shared component. In Google App Engine and AppScale, a *memcached* service is shared between applications by default [37]. We introduce a previously-reported bug in the AppScale platform when using *memcached* [9]. To observe data leakage, we use a Python forum application called *Fofou* [28], which lets clients create forums and post comments.

Misconfiguration. In AppScale, the keys that are used for *memcached* lookups are prefixed with an application identifier. We modify the code of the `_GetKey` function in `/AppServer/google/appengine/api/memcache/memcache_distributed.py` to remove this application identifier.

Tagging and deployment. We mark the form field "title" in Fofou for tagging and deploy two separate instances, instance A and instance B, of the application on our AppScale deployment. One client of each application instance sends a request to create a forum. The client of instance B then requests a list of forum topics and is able to see the forum created in instance A.

Detection. Instance A only has two identical tag entries, an ingress and a non-ingress one; the latter corresponds to the data being sent to the container that runs *memcached*. It is able to decrypt both tags.

Instance B has six tag entries in its log: (1) an ingress entry for its own data; (2) its own data propagating to the *memcached* container; (3) instance A's data retrieved from the *memcached* container; (4) its own data retrieved from the *memcached* container; (5) instance A's data sent to the client; and (6) its own data sent to the client. It is unable to decrypt tags (3) and (5), and has no ingress entries for them. It forwards them to instance A, which can decrypt them, indicating a data leak.

Incorrect request routing. When applications are scaled in OpenShift, the *haproxy* [45] service is used to load-balance between multiple containers serving the same application. We introduce an error in the configuration file read by *haproxy* to decide how to route requests between different containers.

Misconfiguration. We deploy two instances, A and B, of WordPress and manually scale them to use two containers for handling client requests. The *haproxy* of instance A load-balances between containers 1 and 2, and the *haproxy* of instance B load-balances between containers 3 and 4. We then change the configuration file of instance A in `~/haproxy/conf/haproxy.cfg` to replace the IP address of container 2 with that of container 4.

Tagging and deployment. We select the form fields "post-title" and "content" for tagging. Since the load-balancing is performed in a round-robin fashion, the second client request to instance A is routed to container 4 serving instance B.

Detection. Since the instances are scaled, there are two tag logs per instance, corresponding to the two containers serving the instances. Containers 1 and 3 have the ingress monitors for instances A and B, respectively, because they run the *haproxy* services. Due to the leak, instance B sees two entries in the tag log of container 4, caused by the data entering container 4 and leaving container 4 to go to the MySQL container. Instance B is unable to decrypt these two tags and has no ingress entries for them. Instance A decrypts them successfully.

Incomplete data deletion. In this scenario, a system administrator inadvertently forgets to delete and recreate a file system. Instead the file system, which used to belong to a previous tenant, is assigned to a new tenant. Deployed applications rely on specifically named files such as `app_`*{id_number}*`.txt`. If application identifiers are

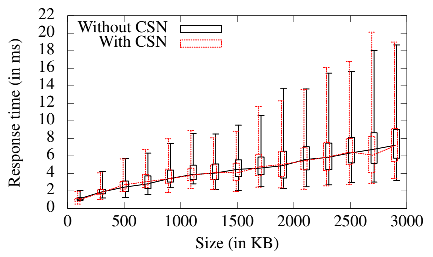

Figure 5: Response times for different sizes of HTTP response data

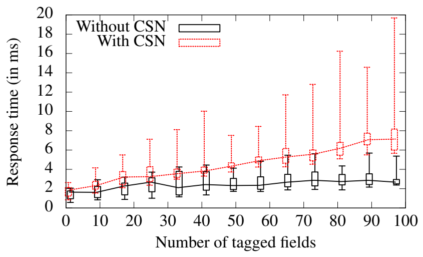

Figure 6: Response time for different numbers of tagged fields

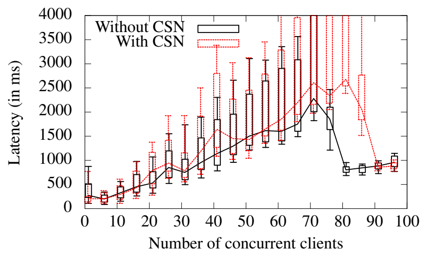

Figure 7: Response time with different number of concurrent clients

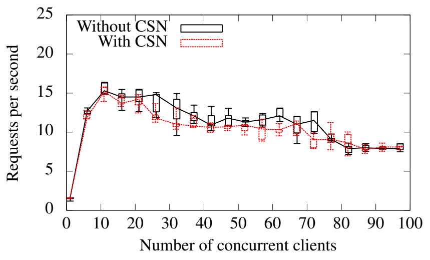

Figure 8: Request throughput with different number of concurrent clients

tenant-specific, a new application may thus read a file that belonged to the previous tenant.

Misconfiguration. Consider two instances, A and B, of the minimal PHP application discussed earlier, both deployed with the same application identifier 100. After sending a client request to instance A, the data is stored in `/tmp/app100.txt`. Next, the file system containing this file is exposed to instance B.

Tagging and deployment. Both instances add tags to the form field. When a client of instance B makes a request, it receives data from instance A.

Detection. Instance A has a tag log with two entries, which it can decrypt. Instance B has three tag entries, one of which is the tag that corresponds to instance A's data. Since instance B does not have an ingress entry for it, it constitutes a data leak.

5.3 Performance overhead

Next we want to evaluate the performance impact of CSN with a realistic deployment of the the WordPress [92] content management system and using micro-benchmarks. Our results show that, even under a significant concurrent request load, the performance reduction due to CSN is low: the reduction in the request throughput even for a large number of concurrent users is less than 10%.

Request throughput. We explore the performance impact that CSN has on request throughput with a realistic web application. We use WordPress [92] version 3.7.1, a widely-used open-source content management system that uses the Apache HTTP server for serving requests and MySQL as its storage backend. We deploy two concurrent instances of WordPress on our OpenShift installation: one with CSN and one without. For the CSN version, we modify OpenShift to attach monitors automatically to the web server and database containers. All incoming and outgoing traffic for the application components is thus monitored for security tags.

For a realistic client workload, we use *Apache JMeter* [7] to generate HTTP requests based on the following workflow. Each client:

1. visits the main page and goes to the log-in page;
2. logs in and is redirected to the main admin page;
3. goes to the "New post" page and creates a new post;
4. visits the new post and adds a comment;
5. goes to the "Post overview" admin page, deletes the post and logs out.

We select three text fields that are part of the above workflow to be extended with security tags: the post title, the post text and the comment text. The number of concurrent clients is increased over time—adding a new client every 5 seconds. Each client performs the above sequence of actions in a loop. The test results are averaged for 5 of these runs for two experiments—one with CSN deployed and one without.

Fig. 7 shows the response times as we increase the number of concurrent clients. The results show that there is little difference between the performance of both instances of WordPress: even with a high number of concurrent clients, the changes in response time for the WordPress instance with CSN appear to be well within the variance caused by other factors in the network and web server environment. With more than 70 clients, the deployment becomes overloaded, and the web server carries out admission control, resulting in rejected requests.

Fig. 8 shows the request throughput as a function of the number of concurrent clients. We can see that the throughput for the instance with CSN is only slightly lower than for the one without CSN. With more than 15 clients, the throughput of WordPress—irrespective of CSN–drops, and we speculate that WordPress begins to experience contention. Above 70 clients, requests are rejected due to admission control. We conclude that, in practice, a deployment is likely to scale out its infrastructure before the CSN overhead would become a problem.

Size of HTTP response data. For the micro-benchmarks, we deploy *lighttpd* [53], a lightweight web server on our local CloudStack deployment, as described in §5.1. We use *ApacheBench* [6], an HTTP performance measurement tool, to generate a request workload and record response times. First we evaluate the overhead of running CSN to monitor for security tags. We use ApacheBench to generate requests for files varying from 100 KB to 3 MB, which are served by lighttpd. None of the data contains security tags.

Fig. 5 shows the median response time for varying sizes of HTTP responses with and without CSN. CSN introduces negligible overhead in this scenario: even for large responses of 3 MB, the change in the median response time is significantly less than the variance due to network effects. Since the CSN monitors passively observe socket streams, their performance impact is low.

Number of tagged fields. We investigate how the number of observed security tags affects the response time of HTTP requests. We deploy a *FastCGI* script [27] on the web server that accepts an HTTP `POST` request and returns its form-field parameters and values in the HTTP response. The field values consist of random 50-byte strings. With CSN, each form field contains a security tag that is 357 bytes in size.

Fig. 6 shows the change in response times as the number of tagged fields increases. Even for an unrealistically large number of close to 100 tagged fields, the median response increases only by approximately 4.2 ms. The observed increase in latency is due to two effects: (i) each tag is seen twice by the monitor, which must process and record it in the tag log; and (ii) the relative size of tags compared to the rest of the field data is large—for larger field sizes, the processing of the tags is amortised as part of the overall cost of request handling.

6. RELATED WORK

The goal of CSN of improving tenant's trust of cloud platforms is shared by work on remote attestation and accountability. Its tag-based approach to monitoring data flows is related to information flow control techniques, encryption and digital watermarking. We now survey this related work in more detail.

Remote attestation enables users to verify the precise version of software installed on a remote machine. Most remote attestation systems rely on cryptographic hardware support in the form of a *Trusted Platform Module (TPM)* [90] to securely bootstrap the infrastructure [79]. Proposals for software-based attestation target simple devices [86] or cannot be used over wide-area networks [85].

Remote attestation has several drawbacks as a mechanism for preventing data leakage. First, it hinders the management flexibility of cloud providers, making it harder to apply software upgrades, patches or configuration changes [44]. Some approaches such as Excalibur [80] attempt to overcome this issue by using high-level attribute-based attestation instead of hashes, but this limits a tenant's ability to catch misconfigurations. In contrast, CSN does not interfere with current management practices of cloud providers.

Second, it requires cloud providers to expose internals of their infrastructure. For commercial or security reasons, cloud providers are reluctant to do so [80, 78]. EVE [50] is an attestation-like approach for verifying the execution of cloud-hosted web applications that avoids these concerns. It models web applications as object stores and uses a collaborative probing mechanism to detect intra-tenant inconsistencies, which is different from our focus on inter-tenant data leakage. In general, CSN avoids disclosing exact configuration details.

Other work has used attestation to improve security in virtualised environments [33, 70]. An important issue is to minimise vulnerabilities in the *trusted computing base (TCB)*. Proposed techniques include exploiting new hardware protection capabilities in AMD and Intel processors [56], reducing the size of the TCB [99] and narrowing the VM management interface [61]. This helps prevent malicious users from compromising the security of the cloud platform, but is orthogonal to our goal of detecting errant data leakage.

Accountability. CSN can be viewed as a mechanism for improving the accountability [94, 95] of cloud platforms with respect to data leakage. Early work on accountability investigated application-specific techniques for ensuring integrity of network storage services [96] and peer-to-peer systems [57].

PeerReview [43] is a technique for ensuring accountability in distributed systems. By keeping a secure record of the messages exchanged between components, PeerReview detects when a component's behaviour deviates from that of a given reference implementation. However, PeerReview assumes deterministic components, which is challenging in cloud environments.

To work around this limitation, Haeberlen et al. [42] propose *accountable virtual machines (AVMs)*. AVMs record all external communication and non-deterministic input of applications in a tamper-evident log. To check for misbehaviour, auditors replay inputs from the log on a reference VM—a heavyweight process.The same authors argue that accountability in cloud computing is important, but note that existing techniques do not address data leakage [41].

Information flow control (IFC) techniques have been employed in a variety of contexts, including at the OS [97, 52], middleware [58] and language levels [93, 67], databases [82] and in distributed settings [98].

SilverLine [60] is an IFC system for ensuring data and network isolation in cloud services. Tenant-provided policy is used to label data, which SilverLine leverages to provide strong enforcement guarantees in addition to data-flow tracking. However, SilverLine relies on a modified OS kernel, does not present performance results and does not allow tenants to check whether it is working correctly. In general, CSN differs from work on IFC in that it attempts to *detect* data leakage but not *prevent* it. It therefore has lower runtime overhead and avoids substantial modification of applications, OS kernels, databases or hypervisors. It also permits tenants to check the operation of monitors.

Closer to our approach is CloudFence [68], which uses binary byte-level taint tracking and an append-only audit log to monitor the propagation of user data in a cloud environment. Finer-granularity tracking means CloudFence can detect a wider range of data leaks than CSN but at the cost of a higher performance penalty: CloudFence reduces the request throughput for a CPU-bound web application by an order of magnitude, compared to the negligible performance impact of CSN.

Ganjali and Li [32] use IFC to deter malicious administrators in cloud environments. Their system records all data flows between VMs and administrators in a secure audit log. Unlike CSN, they focus on detecting malicious behaviour and not accidental data leaks.

Shu and Yao [87] propose a fuzzy fingerprinting technique to detect undesirable data flows without requiring a plaintext database of sensitive data. CSN differs in its goal of detecting inter-tenant data leakage—e.g. it must distinguish between genuine data leaks and data of different tenants that is coincidentally the same.

Data Marking techniques for multimedia [46] and relational data [4] involve perturbing data in a way that is difficult to detect, without affecting application semantics. In contrast CSN does not try to hide the presence of its markers from users.

Encryption Functionally encryptable data can be extracted automatically and encrypted without impacting application semantics as demonstrated in Silverline et al. [76]. However, if data remains sensitive for years, even leaks of encrypted data may be undesirable [88]. Since data in CSN remains unencrypted, there is less overhead, and applications can perform additional operations on tagged data such as keyword search.

7. CONCLUSIONS

In complex cloud environments, data leakage due to accidental misconfigurations and bugs in cloud platforms increases. Prior proposals to prevent data leakage constrain cloud providers in their management activities or introduce a large performance overhead.

Instead, we described *CloudSafetyNet (CSN)*, a lightweight approach for monitoring data propagation in PaaS clouds in order to discover data leakage between tenants. CSN adds security tags to a subset of all client HTTP request fields and uses a set of monitors to observe the propagation of tags, discovering data leakage. CSN can increase the confidence of tenants that their data remains isolated and is compatible with a wide range of PaaS platforms while requiring only minimal changes to existing cloud environments.

8. ACKNOWLEDGEMENT

This work was supported by grant EP/K008129/1("CloudSafetyNet: End-to-End Application Security in the Cloud") from the UK Engineering and Physical Sciences Research Council (EPSRC).

9. REFERENCES

[1] National vulnerability database. `http://nvd.nist.gov/`.

[2] Open source vulnerability database. `http://osvdb.org/`.

[3] Secunia. `http://secunia.com/community/advisories/`.

[4] AGRAWAL, R., AND KIERNAN, J. Watermarking relational databases. In *VLDB* (2002).

[5] AMAZON. AWS security center. `http://aws.amazon.com/elasticloadbalancing/`, 2014.

[6] APACHEBENCH. Apache HTTP server benchmarking tool. `http://httpd.apache.org/docs/2.2/programs/ab.html`.

[7] APACHEJMETER. `https://jmeter.apache.org/`.

[8] APPARMOUR. `http://wiki.apparmour.net/index.php/Documentation/`.

[9] APPSCALE. Add namespacing to Memcache. `http://code.google.com/p/appscale/issues/detail?id=172`, 2010.

[10] APPSCALE. Change AppServer to prevent file system access. `http://code.google.com/p/appscale/issues/detail?id=167`, 2010.

[11] APPSCALE. Autoscaling in AppScale. `http://www.appscale.com/blog/2013/11/02/autoscaling-in-appscale/`, 2013.

[12] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *OSDI* (1999).

[13] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970).

[14] CHOHAN, N., BUNCH, C., PANG, S., ET AL. Appscale: Scalable and open appengine application development and deployment. In *Cloud Computing*. Springer, 2010.

[15] CLOUD SECURITY ALLIANCE. Cloud computing vulnerability incidents: A statistical overview. `http://goo.gl/oaQYaH`, 2013.

[16] CLOUDFOUNDRY. `https://www.cloudfoundry.com/`, 2014.

[17] CLOUDSTACK. `http://cloudstack.apache.org/`, 2014.

[18] COMPUTERWEEKLY. Cloud revenues to touch $20bn by 2016 with PaaS as the fastest growing segment. `http://goo.gl/spqRMF`, 2013.

[19] DAWSON, G., AND DAWSON, M. Introduction to Java Multitenancy. `https://www.ibm.com/developerworks/java/library/j-multitenant-java/`, 2013.

[20] DEIS. `http://deis.io`, 2014.

[21] DIGITALOCEAN. Digital Ocean API is not told to scrub (securely delete) VM on destroy. `https://github.com/fog/fog/issues/2525`, 2013.

[22] DOCKER. `https://www.docker.io/`, 2014.

[23] DOKKU. `https://github.com/progrium/dokku`, 2014.

[24] DOM. The Document Object Model. `http://www.w3.org/`, 2014.

[25] DROPBOX. The Dropbox Blog: Yesterday's Authentication Bug. `https://blog.dropbox.com/2011/06/yesterdays-authentication-bug/`, 2011.

[26] EDGE, J. LSS: Secure Linux containers. `https://lwn.net/Articles/515034/`, 2012.

[27] FASTCGI. `http://www.fastcgi.com/`, 2014.

[28] FOFOU. `http://blog.kowalczyk.info/software/fofou/`.

[29] FOUNDATION, A. S. Apache module mod_so, LoadFile directive. `http://httpd.apache.org/docs/2.2/mod/mod_so.html`, 2014.

[30] FSISAC. Financial services information sharing and analysis center. `https://www.fsisac.com`, 2014.

[31] GAL-OR, E., AND GHOSE, A. The economic incentives for sharing security information. *Information Systems Research 16*, 2 (2005).

[32] GANJALI, A., AND LIE, D. Auditing cloud management using information flow tracking. In *ACM workshop on Scalable trusted computing* (2012).

[33] GARFINKEL, T., PFAFF, B., CHOW, J., ET AL. Terra: A virtual machine-based platform for trusted computing. *ACM SIGOPS Operating Systems Review 37* (2003).

[34] GARRETT, J. J. Ajax: A new approach to web applications. `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/`, 2005.

[35] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC* (2009).

[36] GOOGLE. Google AppEngine. `https://developers.google.com/appengine/`, 2014.

[37] GOOGLE. Google AppEngine Memcache Java API overview. `https://developers.google.com/appengine/docs/java/memcache/`, 2014.

[38] GOOGLE. Implementing multitenancy using namespaces. `https://developers.google.com/appengine/docs/java/multitenancy/multitenancy`, 2014.

[39] GOOGLE. The JRE class white list. `https://developers.google.com/appengine/docs/java/jrewhitelist`, 2014.

[40] GOOGLE. Storing data in Google AppEngine. `https://developers.google.com/appengine/docs/python/storage`, 2014.

[41] HAEBERLEN, A. A case for the accountable cloud. *ACM SIGOPS Operating Systems Review 44*, 2 (2010).

[42] HAEBERLEN, A., ADITYA, P., RODRIGUES, R., AND DRUSCHEL, P. Accountable virtual machines. *OSDI* (2010).

[43] HAEBERLEN, A., KOUZNETSOV, P., AND DRUSCHEL, P. PeerReview: practical accountability for distributed systems. *ACM SIGOPS Operating Systems Review 41*, 6 (2007).

[44] HALDAR, V., CHANDRA, D., AND FRANZ, M. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium* (2004).

[45] HAPROXY. `http://haproxy.1wt.eu`, 2014.

[46] HARTUNG, F., AND KUTTER, M. Multimedia watermarking techniques. *Proceedings of the IEEE 87*, 7 (1999).

[47] HAYDEN, M. `http://stopdisablingselinux.com/`, 2014.

[48] IDG ENTERPRISE. Cloud computing survey. `http://goo.gl/hGDus0`, 2012.

[49] ITISAC. Information technology information sharing and analysis center. `http://www.it-isac.org`, 2014.

[50] JANA, S., AND SHMATIKOV, V. EVE: Verifying correct execution of cloud-hosted web applications. In *HotCloud* (2011).

[51] KERRISK, M. Namespaces in operation, part 5: User namespaces. `http://lwn.net/Articles/532593/`, 2013.

[52] KROHN, M., YIP, A., BRODSKY, M., ET AL. Information flow control for standard OS abstractions. *ACM SIGOPS Operating Systems Review 41*, 6 (2007).

[53] LIGHTTPD. `http://www.lighttpd.net`, 2014.

[54] LXC. Linux Containers. `http://linuxcontainers.org`, 2014.

[55] Marnix Dekker, Dimitra Liveri, M. L. Incident reporting for cloud computing. http://www.enisa.europa.eu/activities/Resilience-and-CIIP/cloud-computing/incident-reporting-for-cloud-computing, 2010.

[56] McCune, J. M., Li, Y., Qu, N., et al. TrustVisor: efficient TCB reduction and attestation. In *IEEE Security and Privacy (S&P)* (2010).

[57] Michalakis, N., Soulé, R., and Grimm, R. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *NSDI* (2007).

[58] Migliavacca, M., Papagiannis, I., Eyers, D. M., Shand, B., Bacon, J., and Pietzuch, P. Distributed middleware enforcement of event flow security policy. In *Middleware* (2010).

[59] Morgan, A. G., and Kukuk, T. The Linux-PAM System Administrator's Guide. http://www.linux-pam.org/Linux-PAM-html/Linux-PAM%5FSAG.html, 2010.

[60] Mundada, Y., Ramachandran, A., and Feamster, N. SilverLine: data and network isolation for cloud services. In *HotCloud* (2011).

[61] Murray, D. G., Milos, G., and Hand, S. Improving Xen security through disaggregation. In *VEE* (2008).

[62] OpenShift. https://www.openshift.com/, 2014.

[63] OpenShift. External load balancing in OpenShift. http://goo.gl/aLZVQN, 2014.

[64] OpenShift. The future of OpenShift and Docker containers. https://www.openshift.com/blogs/the-future-of-openshift-and-docker-containers, 2014.

[65] OpenShift. OpenShift Gears. http://openshift.github.io/documentation/oo_system_architecture_guide.html, 2014.

[66] OpenVZ. http://openvz.org, 2014.

[67] Papagiannis, I., Migliavacca, M., and Pietzuch, P. PHP Aspis: using partial taint tracking to protect against injection attacks. In *USENIX WebApps* (2011).

[68] Pappas, V., Kemerlis, V. P., Zavou, A., et al. CloudFence: data flow tracking as a cloud service. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2013.

[69] PCWorld. Microsoft BPOS service hit with data breach. http://goo.gl/xLuPAZ, 2010.

[70] Perez, R., Sailer, R., and van Doorn, L. vTPM: virtualizing the trusted platform module. In *USENIX Security* (2006).

[71] Peter Loscocco, N. S. A. Integrating flexible support for security policies into the linux operating system. In *USENIX ATC* (2001).

[72] Pham, T. The importance of information-sharing in countering security threats. http://goo.gl/Iz4HmB, 2014.

[73] pidCrypt. https://www.pidder.de/pidcrypt, 2014.

[74] Ponemon Institute. Security of cloud computing providers study. http://goo.gl/SkT3Jx, 2011.

[75] Popa, R. A., Redfield, C., Zeldovich, N., and Balakrishnan, H. Cryptdb: protecting confidentiality with encrypted query processing. In *SOSP* (2011).

[76] Puttaswamy, K. P., Kruegel, C., and Zhao, B. Y. Silverline: toward data confidentiality in storage-intensive cloud applications. In *ACM Symposium on Cloud Computing* (2011).

[77] Reed, J. Following incidents into the cloud. http://www.sans.org/reading-room/whitepapers/incident/incidents-cloud-33619, 2010.

[78] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS* (2009).

[79] Sailer, R., Zhang, X., Jaeger, T., and Van Doorn, L. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security* (2004), vol. 13.

[80] Santos, N., Rodrigues, R., Gummadi, K. P., and Saroiu, S. Policy-sealed data: A new abstraction for building trusted cloud services. In *USENIX Security* (2012).

[81] Schiffman, J., Moyer, T., Vijayakumar, H., et al. Seeding clouds with trust anchors. In *ACM workshop on Cloud computing security* (2010).

[82] Schultz, D., and Liskov, B. IFDB: decentralized information flow control for databases. In *EuroSys* (2013).

[83] Schwartz, E. J., Avgerinos, T., and Brumley, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Security and Privacy (S&P)* (2010).

[84] SELinux. Oddjob can't work. https://www.openshift.com/forums/openshift/oddjob-cant-work, 2012.

[85] Seshadri, A., Luk, M., Shi, E., et al. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *ACM SIGOPS Operating Systems Review 39*, 5 (2005).

[86] Seshadri, A., Perrig, A., Van Doorn, L., and Khosla, P. Swatt: Software-based attestation for embedded devices. In *IEEE Security and Privacy (S&P)* (2004).

[87] Shu, X., and Yao, D. D. Data leak detection as a service. In *Security and Privacy in Communication Networks*. Springer, 2013.

[88] Singh, J., Eyers, D. M., and Bacon, J. Disclosure control in multi-domain publish/subscribe systems. In *DEBS* (2011).

[89] Trusted Computing Group. Collaborative defense. http://goo.gl/3A7Ke8, 2013.

[90] Trusted Computing Group. Trusted Platform Module (TPM). http://www.trustedcomputinggroup.org/, 2014.

[91] W Kuan Hon, C. M., and Walden, I. *Cloud Computing Law*. OUP Oxford, 2013, ch. Negotiated Contracts for Cloud Services.

[92] WordPress. https://www.wordpress.com, 2014.

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

[94] Yumerefendi, A. R., and Chase, J. S. Trust but verify: accountability for network services. In *ACM SIGOPS European workshop* (2004).

[95] Yumerefendi, A. R., and Chase, J. S. The role of accountability in dependable distributed systems. In *HotDep* (2005).

[96] Yumerefendi, A. R., and Chase, J. S. Strong accountability for network storage. *ACM Transactions on Storage (TOS) 3*, 3 (2007).

[97] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and MaziÃĺres, D. Making information flow explicit in HiStar. In *OSDI* (2006).

[98] Zeldovich, N., Boyd-Wickizer, S., and Mazieres, D. Securing distributed systems with information flow control. In *NSDI* (2008).

[99] Zhang, F., Chen, J., Chen, H., and Zang, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *SOSP* (2011).