CloudFence: Enabling Users to Audit the Use of their Cloud-Resident Data

Vasilis Pappas Vasileios P. Kemerlis Angeliki Zavou Michalis Polychronakis
Angelos D. Keromytis

Network Security Lab
Department of Computer Science
Columbia University, New York, NY, USA
{vpappas, vpk, azavou, mikepo, angelos}@cs.columbia.edu

Abstract

One of the primary concerns of users of cloud-based services and applications is the risk of unauthorized access to their private information. For the common setting in which the infrastructure provider and the online service provider are different, end users have to trust their data to both parties, although they interact solely with the service provider. This paper presents *CloudFence*, a framework that allows users to independently audit the treatment of their private data by third-party online services, through the intervention of the cloud provider that hosts these services.

CloudFence is based on a fine-grained data flow tracking platform exposed by the cloud provider to both developers of cloud-based applications, as well as their users. Besides data auditing for end users, CloudFence allows service providers to confine the use of sensitive data in well-defined domains using data tracking at arbitrary granularity, offering additional protection against inadvertent leaks and unauthorized access. The results of our experimental evaluation with real-world applications, including an e-store platform and a cloud-based backup service, demonstrate that CloudFence requires just a few changes to existing application code, while it can detect and prevent a wide range of security breaches, ranging from data leakage attacks using SQL injection, to personal data disclosure due to missing or erroneously implemented access control checks.

1 Introduction

The multifaceted benefits of cloud computing to both service providers and end users have led to its rapid adoption for the deployment of online services and applications. As businesses and individuals increasingly rely on the cloud, some of their private data is being handled and stored on systems outside of their administrative control. In this setting, data confidentiality becomes a growing concern, especially when taking into account the recent spate of security breaches in major online services [39, 13, 42]. In lack of an alternative option other than not using the service at all, most users eventually trust the service provider for keeping their data safe.

Unfortunately, relying solely on reputable service providers or highly popular services does not mitigate the risk. Most feature-rich cloud-based services are quite complex, and are usually based on the integration of a multitude of existing components, such as web servers, databases, and other software modules. Bugs and vulnerabilities in third-party code, misconfigurations and incorrect assumptions about the interaction between different components, or even simple causes like the careless handling of access credentials, can lead to the accidental exposure of critical data, or leave the system vulnerable to data theft attacks. At the same time, cloud computing encourages rapid application deployment, and time-to-market pressure sometimes makes data security a secondary priority.

Despite the existence of a large body of work on data leakage prevention, detection, and mitigation [23, 51, 49, 19], data breaches still pose an important threat. In this work, we seek to reinforce the confidence of end users for the safety of their data, beyond any assurances offered by the online service. To this end, we propose to give users the ability to *audit* their cloud-resident data through a different—and potentially more trustful—entity than the actual provider of the service. This can be achieved by taking advantage of the multi-party trust relationships that exist in typical cloud environments [10], in which the service provider is different than the provider of the infrastructure on which the service is hosted.

As a step towards this goal, in this paper we present CloudFence, a data flow tracking (DFT) framework for cloud-based applications. CloudFence is offered by cloud hosting providers as a service to their tenants, as well as to the users of the tenants' services. Through a simple API, service providers can easily integrate data

flow tracking in their services, and mark sensitive user data that needs to be protected. End users can then monitor the propagation of their data directly through the cloud hosting provider, ensure that all sensitive data is treated as expected, and spot any deviations. Service providers can also take advantage of data flow tracking for enabling an additional layer of protection against data leaks, by preventing the propagation of marked data beyond a set of specified network and file system locations.

We have implemented CloudFence on top of a fine-grained data flow tracking library based on runtime binary instrumentation. CloudFence is dynamically attached to the processes that comprise a cloud application, such as web servers and databases, including processes that run on different physical or virtual hosts. Cross-application and cross-host tag propagation is handled transparently, without requiring any modifications to application code.

In our current prototype, service providers are responsible for specifying the sources of sensitive data and associating them with each user, as well as for defining the allowed data flow paths and confinement points. This means that we consider service providers as trusted, and willing to integrate CloudFence into their applications, as well as to cooperate with the cloud provider for providing an added-value service to their users. However, integrating CloudFence in a more strict cloud environment, such as Google App Engine, which exposes to developers only well-defined APIs for building their applications, would allow for robust data flow tracking and auditing even in the presence of a malicious service provider.

We evaluated the effectiveness and performance of CloudFence using three real-world applications, and two publicly disclosed data leakage vulnerabilities in two of these applications. CloudFence can be easily integrated in these applications, since it requires the placement of just a few API calls in each case, while it offers effective protection against a wide range of data theft attacks, including SQL injection and arbitrary file read attacks.

The runtime overhead due to dynamic instrumentation in an e-commerce application imposes an average slowdown that ranges between 60–90%, for an intensive scenario with a hundred concurrent clients. Although significant, this overhead is amplified by the dynamic code generation behavior of PHP, which hinders the effective caching of instrumented code blocks, and can be ameliorated using PHP bytecode caching techniques. Still, even under these worst-case conditions, CloudFence remains a practical solution for real-world services.

2 Approach

Users of online services trust the providers of those services to securely handle and protect their personal information. Access credentials, credit card and social security numbers, documents, and other kinds of sensitive data are temporarily or permanently stored in back-end databases and file systems, out of user control. Even when service providers are considered trusted and follow best security practices, unauthorized access to sensitive data remains a plausible threat, e.g., due to vulnerabilities in third-party software components that are part of the service.

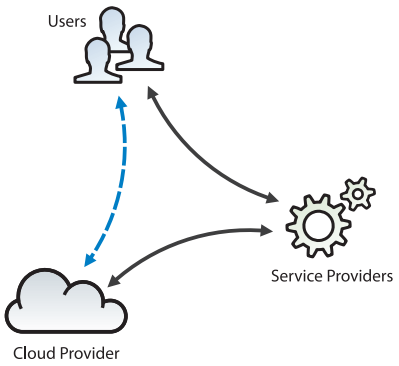

Figure 1: CloudFence allows users to directly audit their cloud-resident data independently through the cloud hosting provider.

For cloud-based applications, service providers in turn place their trust in the cloud infrastructure that hosts their online services. The traditional provider-user relationship is thus transformed into a multi-party system [10], in which end users are usually not aware at all of the existence of the cloud infrastructure provider (unless it is the same entity that also offers the service to end users, as for example is the case with many of the web applications offered by Google or Amazon). In this work we refer to both infrastructure and platform "as a service" (IaaS/PaaS) providers as *cloud providers*. Their infrastructure hosts the applications of *service providers*, which are delivered as services to *end users*.

From the users' perspective, there is an inherent shared responsibility between the cloud and the service providers regarding the security guarantees of the provided service. Despite the fact that end users do not interact directly with the cloud provider, they implicitly trust its infrastructure—the systems in which their data are kept. CloudFence aims to promote and expose this implicit trust for the benefit of all parties, by introducing a *direct* relationship between end users and the cloud provider, as illustrated in Figure 1.

With data flow tracking as the basic underlying mechanism, the cloud provider offers end users the ability of data auditing, allowing them to inspect the audit trail of any sensitive data that was handled as part of a service hosted on the cloud provider's infrastructure. While the

trust relationship between users and service providers is not altered, users get an elevated degree of confidence by being able to independently monitor their private information as it propagates through the cloud. In fact, users are likely to be more willing to trust a large, well known, and highly reputable cloud provider, compared to a lesser-known developer or company, e.g., among the thousands that offer their applications and services through online application distribution platforms.

At the same time, service providers themselves can take advantage of data flow tracking to confine the use of sensitive user data in well-defined network and file system domains, and thus prevent inadvertent leaks or unauthorized data access. Besides protecting user data, a service provider can also take advantage of CloudFence as an additional level of protection for its own digital assets, such as back-end credentials, source code, or configuration files.

Finally, by integrating CloudFence in its infrastructure, the cloud provider offers added value to its tenants and their users, which can potentially lead to a larger customer base. Given the shared responsibility between cloud and service providers regarding the safety of user data, both have an extra incentive to adopt CloudFence as a means of providing an additional level of assurance to their customers.

Threat Model. We assume that service providers are trusted, and integrate CloudFence in their applications to enhance the security of the provided services. This is a typical situation for cloud-based services, since end users always implicitly trust their data to both the service provider and the cloud hosting provider.

Our current implementation is built on top of a user-level data flow tracking system based on runtime binary instrumentation, which is directly integrated into the components of the protected service through an API provided by the cloud provider. Application developers are responsible for specifying the sources of sensitive user input, so that all necessary data is always being marked and tracked appropriately.

Data flow tracking at user-level means that an attacker that gains arbitrary code execution can bypass data tracking and exfiltrate sensitive information without being logged. However, our prototype offers protection against many other classes of attacks that can lead to unauthorized data access, but which do not allow arbitrary code execution. Such attacks include SQL injection, command injection, parameter tampering, directory traversal, and other attacks that are widely seen in the wild. Note that this is a limitation of our current prototype and not of the CloudFence framework in general. An alternative implementation using data flow tracking at the hypervisor level [50] would allow for accurate data tracking even in the case of a fully compromised guest OS. We discuss this issue further in Section 6.

Besides providing protection against external attacks, an equally important goal of CloudFence is to bring into users' attention any unintended data exposure that can lead to unauthorized access. For example, sensitive information can accidentally be recorded in an error log, or be included into a debug memory dump after an application crash.

3 Design

Cloud computing is a paradigm with unique security and privacy aspects. Its key characteristics [41], such as on-demand self-service, ubiquitous network access, and location-independent resource pooling, result into a perimeter-less environment, which poses significant challenges for protecting against unauthorized access to sensitive data.

Instead of trying to fortify the software that operates on private user information [45], or striving to enforce data and network isolation [27], CloudFence adopts a *data-centric* security approach for the cloud setting, by exploiting the implicit trust relationships that exist in cloud computing environments. Specifically, CloudFence builds upon the observation that sensitive data is the valuable aspect that needs to be protected. By providing auditing capabilities at an extremely fine-grained granularity (down to the byte level) and across the whole cloud infrastructure, CloudFence can alert users for confidentiality breaches and information leaks. In the rest of this section, we describe the challenges for integrating data flow tracking into cloud infrastructures, and present a design that addresses these challenges.

3.1 Data Flow Tracking as a Service

Integrating data flow tracking capabilities in a cloud environment is not a trivial task, since many issues have to be taken into consideration. First, the on-demand consolidation of computing elements, which alleviates the costs of over-provisioning via elastic scaling [10], allows service providers to easily "glue" together functionality and content from third-party sources, to build feature-rich applications. For instance, the term *mashup* is colloquially used to refer to web application hybrids that combine services—from potentially untrusted principals—to offer "rich web experience." [1] The benefits of such an agile development and service provisioning approach are numerous, and therefore, it is critical not to interfere

[1] *"Given a choice between dancing pigs and security, users will pick dancing pigs every time."* – Edward Felten & Gary McGraw.

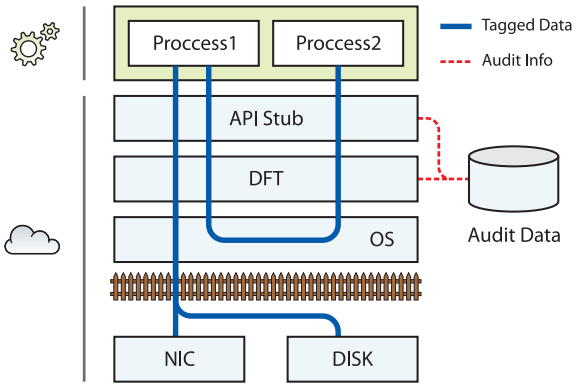

Figure 2: CloudFence architecture.

with that paradigm while enabling sensitive data tracking. We consider this as the *transparent tracking* requirement. The proposed method for data flow tracking should not require intrusive changes, such as manually annotating source code [33] or using custom OSs [48] and modified hypervisors [50], to facilitate incremental deployment and expeditious adoption.

Second, the granularity at which sensitive information is tracked throughout the cloud infrastructure, plays a crucial role in the effectiveness of DFT. Particularly, a service provider can trace data as small as a single byte [30] and provide robust protection against extreme cases of data leakage, or assume a more coarse-grained, and hence error-prone, approach [27]. However, extremely fine-grained DFT comes at a significant performance cost, as tracking logic becomes more intricate (e.g., consider the case of two 32-bit numbers that have only some of their bits marked as sensitive). We consider this as the *fine-grained tracking* requirement, which suggests performing DFT at the appropriate granularity for balancing overhead and accuracy. The scale of choice should be pertinent and without false positive outcomes, even in situations where multi-principal data are fed into a composite application.

Third, given the range of cloud delivery mechanisms with different compositional characteristics (e.g., IaaS, PaaS), it is important to ensure that dynamic collaboration is taken into consideration when performing DFT. The *domain-wide tracking* requirement refers to the precise monitoring of data flows that result from on-demand synthesis, and take place during the transient interactions among application components. In particular, the DFT method should be able to accurately track sensitive data that are delivered to application modules beyond the process boundary. Examples include intra-host application elements that communicate through the file system or OS-level IPC, or consolidated application components running on remote endpoints.

3.2 Architecture Overview

The design of CloudFence addresses the challenges presented in the previous section, while providing fine-grained auditing capabilities for cloud-based services. Note that for the rest of our discussion, we assume that the service provider relies on an IaaS delivery mechanism, which represents an extreme case of data flow tracking complexity. However, CloudFence is by no means limited only to this setting, and can be seamlessly employed in PaaS and SaaS setups.

CloudFence consists of two main components, which we summarize below and describe in detail in the following sections.

- **DFT** The data flow tracking component is the nucleus of CloudFence, and an essential part of our architecture. It performs fine-grained, byte-level data flow tracking without requiring any modification to applications or the underlying OS. Briefly, our DFT component is application *agnostic*; it uses dynamic binary instrumentation (DBI) for retrofitting the data flow tracking logic into unmodified binaries dynamically, at runtime, and supports tracking across processes running on the same or remote hosts. It *piggybacks* tags on the data exchanged through IPC mechanisms or network I/O channels, and transparently handles (un)marshalling, and keeps persistent tag information for marked data written to files.

- **API Stub** The CloudFence API allows service providers to *tag* (i.e., attach metadata information) on sensitive user data that enters their applications. Note that CloudFence does not require application modifications as far as data tracking is concerned (this is handled transparently by DFT). However, it requires slight changes to application code for labeling any sensitive information.

Figure 2 illustrates the architecture of CloudFence. The two processes in the upper part of the figure represent components of a consolidated application, while the rest of the components are part of the cloud provider's infrastructure. Recall that we assume an IaaS delivery mechanism, and this particular example both processes run on the same (virtual) host.

The DFT component is attached to all processes of a composite application, acting as a *reference monitor* [5] that, loosely stated, tracks every byte transfer in the process memory, as well as between the process and the OS. The API stub is used directly by the processes to tag sensitive data and convey to CloudFence which byte sequences correspond to chunks of sensitive information.

Data that are tagged as sensitive (denoted by the solid line in the figure) are tracked across all local files, host-

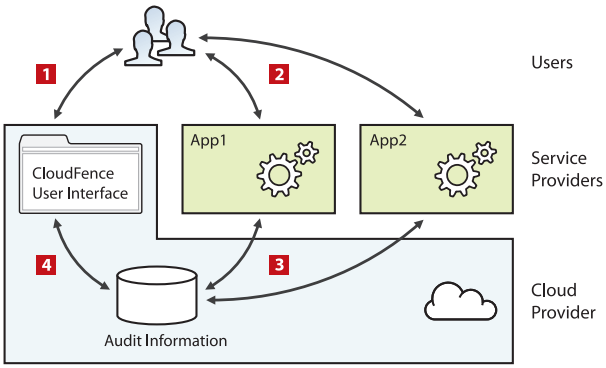

Figure 3: Main interactions among the different parties involved in the use of CloudFence-enabled services.

wide IPC mechanisms, and selected network sockets. Tagged bytes that pass through I/O channels and are written to a cloud storage device, or are transmitted to a remote host via the network, result into an audit message (denoted by the dashed line). Audit information is kept into data store located outside the vicinity of the service provider, and more importantly, operates in an "append-only" fashion for preventing tampering with archived audit trails. The audit messages sent by the DFT component capture leakage events that result from writing private information into files or remote endpoints, while messages sent by the API stub are mostly for initiating the audit process.

Putting it all together. Figure 3 depicts the main interactions among the different parties that are involved in a CloudFence-enabled service.

Initially, users aregister with the cloud provider (1), and acquire a universally unique audit identifier (UUAID). Then, they use the online services offered by various service providers (hosted on the same cloud provider) by providing the UUAID acquired from the previous step (2). The actual mechanism used for conveying the UUAID of each user to CloudFence is not addressed in this work, and is left for future consideration. As an example, the service provider can either request from the users to provide their UUAID during the sign up process with the corresponding application,[2] or in case a cloud-wide identity management system is in place (e.g., a single sign-on system like OpenID), the application can access the respective UUID transparently by requesting it directly from the cloud provider after the user has successfully authenticated. Sensitive data is tagged by the service provider with the supplied user UUAID, and is tracked throughout the cloud infrastructure, while audit information is gathered and stored at the cloud provider (3). At any time, users can monitor the audit trails of their data directly through the cloud provider (4).

[2]Special consideration needs to be taken in that case, so as to deal with illegitimate users that supply UUAIDs that belong to others.

3.3 DFT Component

Data flow tracking is a well-researched area that builds upon Denning's seminal work on secure information flow [16]. In the past, it has been mainly used for detecting unauthorized data usage and enforcing safe information flows, and thus, it is known as information flow tracking [40] and information flow control [28].

3.3.1 Method Synopsis

Fundamentally, the DFT process is characterized by three aspects that we summarize below.

- **Data sources** Data sources are entry points for interesting data. Specifically, they are program or memory locations at which data of concern enter a system. Examples include functions, system calls, and shared memory segments. Data originating from a data source is tagged either entirely, or partially based on an application-specific policy.
- **Data tracking** During the execution of a program, tagged data are tracked and their tags get propagated in accordance to the semantics of the executed instructions. For instance, whenever tagged data are copied, modified, or combined, their respective tags are accordingly propagated to the end result.
- **Data sinks** Data sinks are also program or memory locations where assertions can be made regarding the presence of tagged data, for inspecting or enforcing specific data flows. For example, tagged data may not be allowed in certain memory pages or function arguments.

From an operational perspective, DFT has two requirements. First, it requires extra memory for keeping the data tags of a program, and second, the program itself needs to be extended with tag propagation logic, as well as data tagging and checking logic at data sources and sinks. The code that implements this functionality can be incorporated in source code with source-to-source code transformation [45], retrofitted dynamically into unmodified binaries using dynamic binary instrumentation (DBI) [31, 34], integrated into virtual machine monitors (VMMs) [21] and full system emulators [11, 32, 7], or implemented in hardware [40, 43].

3.3.2 libdft: A Fast and Reusable DFT Framework for Commodity Systems

CloudFence is based on libdft, a homegrown framework [24] in the form of a shared library, for seamless

data flow tracking within a single process. The novel design of the library offers low performance overhead, versatility, and practicality, and allows the development of DFT-aware tools in an effortless manner, thus conforming to the challenges presented earlier in Section 3.1. At the same time, libdft performs *fine-grained*, *dynamic* DFT without requiring any modification to applications or the underlying OS, and transparently handles multi-process and multithreaded programs.

The tracking process is performed dynamically by employing Intel's Pin DBI tool [26]. Pin injects a tiny user-level VMM inside already running processes, or in new processes at startup, and provides an extensive API that libdft uses for inspecting and modifying the process binary at the instruction level. Note that instrumenting at the process level, and not at the hypervisor, has not only performance benefits, but also alleviates any "semantic gap" issues due to VMM introspection. In particular, libdft uses Pin to analyze all instructions that move or combine data for determining data dependencies. Then, based on the discovered dependencies, it instruments the program code by injecting the respective tag propagation logic before the corresponding instructions. Both the original and the additional instrumentation code are re-translated using Pin's just-in-time compiler, to generate the DFT-enabled code that will actually run. However, this process is performed only once, right before executing a previously unseen sequence of instructions, and the instrumented code is placed into a code cache to avoid paying the translation cost multiple times.

3.4 CloudFence API

The API of CloudFence consists of three calls with the following C prototypes:

```
int                             /* success/failure     */
add_tag(const void *buf,        /* starting address    */
        size_t len,             /* length              */
        const char *label)      /* label (byte tag)    */

int                             /* success/failure     */
del_tag(const void *buf,        /* starting address    */
        size_t len)             /* length              */

int                             /* success/failure     */
copy_tag(const void *dst,       /* destination address */
         const void *src,       /* source address      */
         size_t len)            /* length              */
```

The `add_tag` function is used by service providers for tagging, i.e., associating a byte label, to every byte in the address range `[&buf, &buf + len]`, while `del_tag` is used for removing previous associations and unlabeling data. The `copy_tag` function is similar to `libc`'s `memcpy`, but instead of copying data, it propagates the tag information maintained by libdft for the data in `[&src, &src + len]` to `[&dst, &dst + len]`. The functionality is necessary for aiding the service provider in dealing with cases of unintended unlabeling, also known as *whitewashing*. We further discuss this issue in Section 6.

To support higher level languages, which are commonly used in web applications, appropriate wrappers can easily be implemented. Specifically, for some of the applications used in our evaluation, we developed a PHP extension that gets two string arguments (other types can be supported likewise) and internally calls the low-level C functions exported by the CloudFence API.

4 Implementation

In this section, we describe the implementation details of the components that comprise CloudFence. From a high-level view, most of CloudFence's functionality is built on top of libdft, except for the CloudFence user interface, which is a web application coupled with a back-end database that users can use for auditing their data.

Specifically, we used libdft v3.14, which, as mentioned in Section 3.3.2, provides support for transparent tag propagation and an API for assigning and manipulating tags on memory-resident data, as well as for hooking system calls or library functions. The current implementation of libdft performs byte-level data flow tracking, and stores the tag for each data byte in shadow byte—allowing support for eight different tag values per byte (further discussed in Section 6).

4.1 Data Tag Propagation and Persistence

Currently, libdft supports tag propagation within the memory space of a single process, and does not handle the case of data written to files or network sockets. However, to allow for accurate data flow tracking throughout a whole cloud-based application, CloudFence requires persistent data tags, as well as tag propagation across different processes, which may run on different (physical or virtual) hosts (see the domain-wide tracking requirement in Section 3.1). To this end, we have built a layer on top of libdft to support tag propagation across BSD sockets, Unix pipes, files, and shared memory.

4.1.1 BSD Sockets and Unix Pipes

Exchange of tag information over sockets and pipes is handled by embedding all relevant data tags along with the actual data that is being transferred. Maintaining the tag propagation logic completely transparent to existing applications, without modifying them or breaking the semantics of their communication, is the most challenging part of this effort.

In our current prototype, the exchanged tag information consists of a copy of the relevant area of the shadow memory that libdft maintains for the transmitted data.

Consequently, the size of each transmitted message doubles, along with the addition of four bytes, which hold the size of the embedded shadow memory copy. Compressed forms of the tag information could also be used to reduce the data overhead, while slightly increasing the computational overhead.

Synchronous I/O. For synchronous I/O, the approach we followed was to hook the `write`, `send`, and `writev` system calls, using libdft's syscall hooking API, and transmit the tag information before the actual data of the original system call. For `send` and `write`, the additional tag data consist of a copy of the shadow memory area corresponding to the original buffer, along with its four-byte size value. In order to avoid any additional copying of the shadow memory regions in `writev`, we create a shallow copy of the vector structure and just update its pointers to the shadow memory parts of the original buffers. The total length of the shadow vector along with the vector itself are prepended again in the `writev` syscall. Similarly, we hook the `read`, `recv`, and `readv` system calls, and read the tag information before the actual data.

Extra care is taken in case the size of the messages sent is different than the size of the receive buffer—which was often the case during our evaluation. More precisely, every message sent can be received (i) at once, (ii) split in multiple parts, or (iii) interleaved. In the first case, the tag data and the original data are received inside the same receiving operation and we simply have to attach the tag data to the original data on the receiver. For messages that are received through more that one read operation, the receiver initially buffers the tag information. Each time a message part is received, its corresponding tag information is attached to it, until the whole message is received. The most difficult case we should handle is when the send buffer size does not much the receive one. For example, the sender transmits two 500-byte messages back to back, and the receiver reads them using a 200-byte buffer. In this case, the third read operation receives 100 bytes of the original data and 100 bytes of the second message's tag information. This case is handled by changing the length argument of the read operation to match the end of the current message.

Non-blocking I/O. In the case of non-blocking I/O, the above system calls may return a special error code as if the requested operation would block (`EAGAIN`). Keeping the exchange of tag information transparent requires special handling of this type of errors. Specifically, if such an error occurs when trying to read the embedded tag information, control returns immediately to the application, as if its read operation failed. If some, but not all, of the tag data is available, the available part is buffered and CloudFence emulates a "would block" error, as if the read operation would block. Similarly, for write operations, we keep accounting of the relevant shadow memory data that is actually sent, and emulate `EAGAIN` errors until all relevant shadow data has been completely transmitted.

The strategy followed for the write operations assumes that the application always retries to send the same data when it fails with a "would block" error. There are two ways to relax this assumption though. First, trade safety for performance by checking if the file descriptor is ready to write and block until the tag data and the message is sent. The other option is to check if the buffer's address or contents are changed between the failed write operations – requires copying the buffer –, cancel any part of the tag data already sent and retransmit the ones that correspond to the new buffer. We should note here that although there were cases of non-blocking I/O in our evaluation, this assumption always held.

Multiplexed I/O. Regarding multiplexed I/O using the `select`, `poll`, and `epoll` system calls, we chose to trade a small performance overhead in favor of a safer hooking implementation. Before each read or write operation, we block until all tag information is received or sent, similarly to the synchronous I/O scenario. A more robust implementation would be to check if any of the ready-to-read file descriptors are waiting to receive a new message, and attempt to first retrieve its tag information. In case only part of the information is available, we can buffer it, and remove the file descriptor from the returned set of `select` or `poll`, as if it was not ready to be read. However, such an implementation could break application semantics, since the actual intention of the application after a `select` or `poll` invocation is not known in advance, e.g., the application could use `recvmsg`, or not read any data at all.

Finally, in some cases, non-blocking I/O can be combined with multiplexed I/O—a case that we encountered during our evaluation. For instance, this situation arises when an application performs a `connect` system call on a non-blocking socket descriptor, and then passes it on as an argument to `select` or `poll` for monitoring its completion. In that case, `connect` returns a special error code (`EINPROGRESS`) and then the application monitors when the socket descriptor is ready for writing, which implicitly means that `connect` completed. The outcome of `connect` can be checked using the `getsockopt` system call. We handle this case by marking the file descriptors for which `connect` returned `EINPROGRESS`, and then searching for them in the ready-to-write sets of `poll` or `select`. The post-`connect` system call hook is then called for each file descriptor found.

4.1.2 Files

Tag information should persist even when data is written into files, so that these tags can be later retrieved when the same or another process accesses the same file. CloudFence supports persistent tagging of file data by employing shadow files. Whenever a file is opened using one of the `open` or `creat` system calls, we also create a second shadow file in the same path, which has the name of the original file appended with a special suffix. This shadow file is mapped to memory and associated with the original file descriptor.

Whenever a process writes to a file using the `write`, `writev`, or `pwrite` system calls, the tag information of the relevant buffer (or buffers, in case of `writev`) is also written in the appropriate offset of the mapped shadow file. Similarly, after a read operation from a file, using `read`, `readv`, or `pread`, the relevant tag information from the corresponding shadow file is also represented at the destination buffer. Finally, by monitoring all the read and write operations on files, we maintain the size of the shadow files in accordance with the originals.

4.1.3 Shared Memory

Another commonly used inter-process communication mechanism that CloudFence supports, is shared memory. Our current implementation supports shared memory regions allocated using `mmap`, but it can be easily extended to cover POSIX API calls (e.g., `shm_open`) or SysV API calls (e.g., `shmget`). CloudFence hooks calls to `mmap`, and for each shared memory region, it creates a shadow copy to hold libdft's tag information.

The `mmap` function supports two ways to create a shared memory region between two processes: anonymously, or by mapping the same file. In the first case, a process creates an anonymous shared memory region and then forks. The pointer to that region is then inherited by the child processes, so we do not have to take any further actions (i.e., the shadow copy is inherited as well). In the second case, the two processes map the same file by specifying the `MAP_SHARED` flag. Again, another shared memory region for keeping tag information is created, but this time instead of being anonymous, it corresponds to a special file. The path for this file is constructed in the same way as described in the case of shadow files, but with a different suffix.

4.2 Data Flow Domain

In CloudFence, data flow tracking is performed within the boundaries of a well-defined *data flow domain*, according to the components of the online service. Service providers specify the set of programs and network hosts that comprise the service, and data tags propagate throughout all processes and inter-process channels. Whenever some tagged data crosses through the defined boundary, e.g., when a destination file or host does not belong to the specified domain, CloudFence logs the action in the audit database, and, depending on the configuration, may block it. As an example, the domain for the bookmark synchronization application that we used in our evaluation spans two processes: a web server and a SQL server.

CloudFence dynamically instruments all relevant processes by following any spawned child processes, and by default enables tag propagation for any communication through sockets, pipes, or shared memory between them. We should note that tag persistence in files and tag propagation in shared memory is transparent and does not affect processes in which CloudFence has not been enabled, since tag information is separate from the actual data. On the other hand, tag propagation through network sockets embeds tag information in par with the transmitted data, and consequently both communicating processes should be aware of that.

To automate the configuration of tag propagation between processes that exchange data through the network, CloudFence maintains a global registry of active sockets that support it. This is implemented by hooking the `connect` and `accept` system calls of processes in the same domain. Each time a connection is attempted, the initiator's address, e.g., `127.0.0.1:56443`, is recorded in a list of endpoints that support tag propagation. At the same time, the other endpoint's address is queried in the list, and if it exists, this means that both endpoints support it, and consequently tag propagation is enabled for this connection. At the server side, upon a call to `accept`, and before the call actually returns, the server's address is inserted in the list of sockets that support tag propagation, if not already present. After `accept` returns, the client's address is queried in the list, and if it exists, then tag propagation is enabled. Note that using this process, service providers must only specify the processes that comprise the cloud application, and then the rest of the tag propagation logic is determined automatically.

For the simple case in which the whole application is deployed on a single (virtual) host, the registry is stored in shared memory and can be accessed by all the CloudFence stubs. To support tag propagation across different hosts, each host maintains a network-accessible registry as part of the CloudFence stubs. When an instrumented process connects to a remote host within the same domain, it first queries the destination host's registry to determine whether the server supports tag propagation, and then proceeds accordingly.[3]

[3] Another implementation approach we explored was out-of-band

4.3 CloudFence User Interface

For our prototype implementation, we developed a rudimentary web interface through which users can browse through the audit logs that have been generated for the applications hosted on the cloud provider. Each application pushes audit messages to this service that is also accessible by end users. In addition, the interface provides simple user and UUIAD managment. Upon registration, users receive a newly generated UUIAD. This can be later used during the registration in any CloudFence enabled service. In summary, this implementation of the CloudFence user interface follows the simple model described in Section 3.2, in which users manually convey the UUAID to each cloud application.

5 Evaluation

In this section, we evaluate CloudFence in terms of ease of deployment in existing applications, runtime performance, and effectiveness against data leakage attacks, using three real-world applications: an e-commerce framework, a network backup system, and a bookmark synchronization service. Our experimental environment consists of three servers, each equipped with two 2.66GHz quad core Intel Xeon X5500 CPUs and 24GB of RAM, interconnected through a Gigabit switch. To better match a cloud infrastructure environment, two of the servers hosting the CloudFence-enabled applications were running VMWare ESXi v4.1, and all applications were installed in virtual machines. The third server was used to simulate clients and drive the experiments, and was running a 64-bit version of Debian 6. Finally, the guest OS in all virtual machines was 32-bit Debian 6.

5.1 Deploying CloudFence

5.1.1 e-store

The first scenario we consider is an online store hosted on a cloud-based infrastructure. Typically, during a purchase transaction, sensitive information like the credit card number and the recipient's postal and email address are transmitted to the online store, and from there, usually to third-party payment processors. The service provider can incorporate CloudFence in the e-store application to allow users to monitor their sensitive data, as well as to restrict the use of sensitive data within the application's domain.

(OOB) socket data, like urgent data in TCP. Although such an implementation would have been more transparent, it could not be used as OOB data interfere with normal data in the implementation of the `select` syscall in Linux. The BSD implementation of the same syscall does not have this issue.

The developers of the e-store know in advance the entry points of sensitive user data to the application, as well as which processes and hosts should be allowed to have access to this data. For instance, after users input their credit card information through the e-store front end, it should only be accessed by the e-store's processes, e.g., its web and database servers. The only external channel through which it can be legitimately transmitted, is through a connection to the third-party payment processor, i.e., a well-known remote server address, which can be included in a white-list describing what sensitive data from the data flow domain of the application is allowed to be sent there.

The application we chose for this scenario is VirtueMart [1]. VirtueMart is an open source e-commerce framework developed as a Joomla component, and is typically used in PHP/MySQL environments, as both VirtueMart and Joomla are written in PHP. We configured VirtueMart to accept payments only through credit card, and set up electronic payments through the Authorize.Net payment gateway service, using a test account.

To incorporate CloudFence, we had to add just a few lines of code in the registration and checkout phases. More precisely, we added a new input field in the registration form for the user's UUAID, a new column in the user's database table and a few lines of code to store it in the database, along with the user's info. As for the checkout phase, we added a few lines of code in the script that processes the payment information. First, the UUAID is queried from the database, using the user ID from the current session. Then the HTTP POST variable that holds the credit card number is tagged by calling the add_tag API, through a PHP wrapper function. It is important to tag this variable as soon as possible, so as to avoid the leakage of untagged copies.

Finally, the data flow domain of the application is the web server process, the database server process and any processes these two may spawn.

5.1.2 Backup service

The second case study is on hosting a backup service on the cloud, using the Amanda network backup [38, 2]. Amanda is a highly configurable backup software, written in C and Perl, which offers a wide range of authentication mechanisms, like ssh, rhosts, kerberos, etc. and a number of different storage back-ends, such as file system, tape devices, Amazon S3, etc. CloudFence's audit capability provides a way for the users to attest the correct use of their backup data. Users know when their backup client sends data (either because it happens in a fixed schedule, or because their client keeps a log) and when it requests data, a recovery is performed manually. By combining this information with the audit trails pro-

vided by CloudFence, a user can pick up unauthorized accesses.

While keeping mostly the default configuration values, we set up a server and three clients. The server was configured to store the backup data on the file system and authentication was performed using the client's host name (rhosts). As in the previous case, the application's source code had to be changed to incorporate CloudFence. Assuming that such a system is manually configured, the only change was to retrieve the user's UUAID from a host name to UUAID map and tag his data whenever a `SENDBACKUP` command is received. Once tagged, data carry their label through the different components of Amanda (`taper`, `dumper`, `chunker`, etc.) until stored in the file system.

Although the configuration used above fits better to a scenario where a small or medium business outsources its backup service, this is not a limitation. The backup service can also be configured for public use, like DropBox. The only differences in adopting CloudFence in that case, would be to modify the registration phase to include the user's UUAID and the storage of that key.

5.1.3 Bookmark Synchronization

The last use case stems from the ever higher demand for personal data – photos, emails, bookmarks, etc.– synchronization services, since many users have more than one personal electronic devices, such as laptop, smartphone, tablet, etc. The scenario in this case is to host a bookmark synchronization service on the cloud. SiteBar [3] is an online bookmark manager written in PHP, which integrates with many modern browsers. When adding a link to SiteBar, users have the option to set its access level, public or private – the access level can be later edited. As a developer, we want to tag only the private links as sensitive.

The task of incorporating CloudFence in SiteBar was very similar with the first case, as both applications are written in PHP and use MySQL as a database back-end. Especially for the registration phase, the changes where almost identical. On the other hand, changing the source code to tag the sensitive data – private links – was a little bit more elaborate, because the sensitivity level of data dynamically changes. Thus, we need to change the code that adds a link and tag it if it is private. In addition, we need to change the code that edits a link. If the access level changes from private to public, we load the link, remove its tag using del_tag and store it again in the database. Otherwise, we repeat the same steps, but instead of removing the tag, we add it using add_tag. It is essential to update the copy in the database on edit, in order for the change to be persistent.

5.2 Effectiveness

To evaluate the effectiveness of CloudFence, we tested whether it is able to audit illegitimate tagged data accesses, performed as a result of an attack. We exploited two publicly disclosed vulnerabilities of in the studied applications.

The first vulnerability allows authenticated users of SiteBar versions earlier than v3.3.8 to read arbitrary files (`CVE-2007-5694`). This is the result of insufficiently checking a user-supplied value through the `dir` argument, which was used as the base directory for reading language specific files, as shown in the snippet below.

```
sprintf($dir.'/locale/%s/%s',$var1,$var2);
```

Passing a file name ending with the URL-encoded value for the zero byte (`%00`) causes the open system call to ignore any characters after it, and thus reading the supplied file.

```
http://SB_APP/translator.php?download
&dir=/var/lib/mysql/SCHEMA/TABLE.MYD%00
```

We installed SiteBar version 3.3.8 on top of PHP version 5.2.3. Then, we repeatedly read files by exploiting this bug through a web browser in a remote machine. CloudFence was able to report all accesses to data with persistent tags in the filesystem.

Another type of attack that usually leads to information leakage is SQL injection. Passing special crafted values which are used by applications to compose SQL queries can lead to arbitrary SQL command execution. The main cause, again, is the insufficient checking of user input values. To demonstrate the effectiveness of CloudFence on auditing (or even preventing) this type of attacks, we used a real-world vulnerability found in VirtueMart version 1.1.4 [4].

The value of the HTTP GET parameter `order_status_id` was not properly sanitized, thus allowing malicious users to change the SQL SELECT query by using a URL like the one below.

```
http://VM_APP/index.php?option=com_virtuemart
&page=order.order_status_form
&order_status_id=-1' UNION ALL SELECT ...
FROM jos_vm_order_payment where order_id='5
```

Which results in the execution of the following query:

```
SELECT * FROM jos_vm_order WHERE
order_status_id=-1' UNION ALL SELECT ...
FROM jos_vm_order_payment where order_id='5';
```

The later SQL query returns a row from the `jos_vm_order_payment` table, where the credit card numbers are stored, instead of `jos_vm_order`.

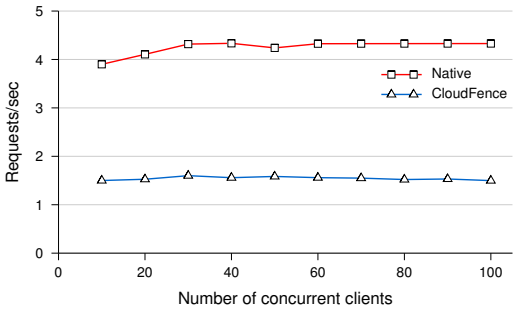

Figure 4: Requests Throughput for VirtueMart installed on a Debian 6, using the default web server configuration (10 pre-forked processes).

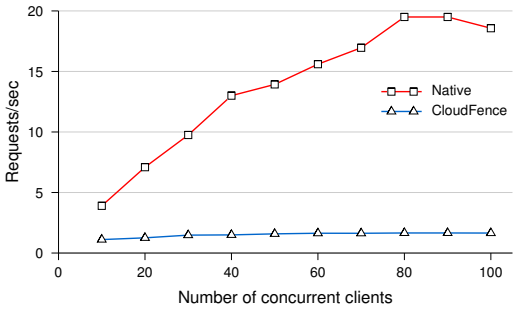

Figure 5: Requests Throughput for VirtueMart installed on a Debian 6, with the web server configured to spawn 100 pre-forked processes.

Similarly, we installed the vulnerable version 1.1.4 of VirtueMart on top of PHP 5.3.3, and tried to steal any stored credit card numbers by exploiting this bug. In all cases, CloudFence identified their exfiltration, as they were tagged as sensitive data upon entry.

5.3 Performance

The last aspect we explored is the performance overhead imposed by CloudFence's fine-grained data flow tracking subsystem, which is a CPU-intensive process due to the use of dynamic binary instrumentation. Our choices for the experimental setup mostly focused on deriving a configuration for a worst case scenario to stress-test our prototype implementation. Among the three applications, we picked VirtueMart, which is the heaviest in terms of code base size and computational overhead. Both VirtueMart and Joomla are written in PHP, which, as an interpreted language, is a worst case for dynamic binary instrumentation frameworks like Pin. The instrumentation code generated by Pin for PHP's dynamically generated code cannot be effectively cached, resulting to poor runtime performance.

VirtueMart was installed in one of the virtual machines of our testbed. The exact versions of the software packages used were the following: VirtueMart v1.1.8, Joomla v1.5.22, Apache v2.2.16, PHP v5.3.3, and MySQL v5.1.49. To generate a realistic and intensive workload, we used a second host connected through a Gigabit switch that emulated typical client requests for placing product purchases. The Gigabit network connection minimizes network latency, increasing this way the stress that we can impose on the server when concurrently emulating multiple user transactions.

Instead of performing the same request over and over, we tried to generate more realistic conditions by replaying complete purchase transactions. Each transaction consists of nine requests: retrieve the front page, login, navigate to the product page for a specific item, add that item in the shopping cart, verify the contents of the shopping cart, checkout, enter payment info, confirm the purchase, and logout. In addition, in each of these requests, the web clients also download any external resources, such as images, scripts, and style files, emulating the behavior of a real browser, without performing any client-side caching. We should stress that VirtueMart was fully configured as in a real production setting, including properly working integration with Authorize.Net for processing credit card payments using a test account.

Using the above setup, we measured the sustained throughput of user requests that the server could handle when processing concurrent transactions from multiple users. As mentioned in Section 3.3.2, Pin performs a slow start when it initially instruments most of the startup code of a process. Instrumented code is then cached, so subsequent executions of the same code blocks are much faster. To prevent Pin's slow start from skewing our results, we "warmed-up" the Apache and MySQL processes by performing a single request to each Apache process. To ensure that each of the warm-up requests is served by a different process in the process pool, we first simultaneously initiate one connection per process, and after all connections have been established, we proceed and send the actual request data.

Figure 4 shows the sustained request throughput of the application for a varying number of concurrent web clients, when VirtueMart is running with and without CloudFence. The request throughput was calculated by dividing the number of requests with the total duration of each experiment. In all runs, each client was configured to perform three end-to-end transactions, so that the number of requests per client remains consistent across all experiments. In this experiment, CloudFence imposes a slowdown of 60%, which is indicative for applications running on top of dynamic binary instrumentation frameworks.

We should note that the server throughput in the native case is not bounded due to limited computation re-

sources, but rather due to the default configuration of Apache, which uses a pool of 10 processes for serving concurrent clients. Thus, to be more precise, CloudFence took advantage of the available cycles and imposed an additional overhead of 60%.

Figure 5 shows the results of the same experiment, but in this case the process pool size of Apache has been increase to 100 processes. In this case, the throughput in the native case is also bounded due to CPU saturation. In the worst case, CloudFence imposes a slowdown of 90% when the number of concurrent users lies between 80–100 users.

Based on these results, in a real deployment of this application, enabling CloudFence would require a twoflod to tenfold increase of the cloud infrastructure resources devoted to the application. However, we should note that the increased overhead of dynamic binary instrumentation due to the dynamically generated PHP code can be effectively mitigated using PHP acceleration techniques based on bytecode caching [37]. These techniques have become mandatory for high-traffic web sites that use interpreted languages. As part of our future work, we plan to investigate the performance overhead of CloudFence under high-end configurations that use bytecode caching.

6 Discussion

The DFT component of CloudFence takes into consideration only cases of explicit data flow, which is in accordance with previous work on the subject [31, 40]. Dytan made some provisions for conditionally handling *implicit* data flows that result from control-flow dependencies, but concluded that while it can be useful in certain domains, it frequently leads to an explosion in the amount of tagged data and to incorrect data dependencies [12]. Despite the fact that ongoing work attempts to address these issues [22], we opted for a design that has zero false positives and tagging pollution. This choice, however, can lead into unintended data whitewashing, whenever the service provider uses a code construct that copies sensitive data using branch statements—which could potentially lead to false negatives.

As an example, consider this code snippet: `if (in == 1) out = 1`. Although the value of `in` is copied to `out`, any tags associated with it are not. During our evaluation, we manually identified a couple of such cases, in AES encryption (used in SSL, MySQL and the Suhosin PHP hardening extension) and Base64 encoding. These cases were easily handled by hooking the corresponding functions and copying the tag information from their source to the target operand using the `copy_tag` function. For example, after encryption using AES, we copy the plaintext tags over those of the ciphertext. Ciphertext carries them until decrypted, when we again copy ciphertext's tags over plaintext's. At the end, the decrypted buffer has the initial tags. The same also holds for Base64 encoding.

Currently, the underlying DFT framework that we use in CloudFence imposes some limitations on the number of different labels that can be associated with each byte of sensitive data. Recall that libdft keeps a shadow byte of tag information, and hence, there is an upper bound on the number of different labels that can exist within the boundary of a single process. We plan to further explore in the future the different tradeoffs between larger tags and performance overhead.

7 Related Work

A common approach for degrading the impact of data leaks is to ensure that important data are always stored in an encrypted form on the remote server [8, 44, 18]. Even though encryption can help with the problem of secure storage in the cloud, it does not solve the security issues of remote data processing in cloud applications. Data must be decrypted before being processed, and then reencrypted, which is a costly process. Added to the inherent latency of the cloud, this can affect endpoint performance. In addition, encryption seems to limit data use, and in particular searching and indexing becomes problematic. Using a homomorphic encryption scheme [20], it is possible to perform certain operations directly on the encrypted data. However, its computation cost is for now prohibitive for real-world applications.

Information flow tracking is another approach for protecting against information leakage. While there is a large body of research focusing on information leakage prevention within a process [12, 31, 34, 51] or a single host [14, 32, 46], it was not until recently that interest has risen for efficient cross-host taint propagation systems [25, 15, 50, 6, 17]. Most of these techniques are more problem-specific, and therefore it would be difficult to adapt them for use in other contexts. For instance, DBTaint [15] is targeting taint information flow tracking specifically for databases. System tomography [29], which also looks into the concept of propagating taint information remotely, builds on the QEMU emulator, which incurs a prohibitively large runtime overhead. Finally, Neon [50] also requires modifications in the underlying system to perform dynamic taint tracking. It uses a modified NFS server for handling the initial tainting, and utilizes a network-filter for monitoring the tainted packets to and from the server.

Jif [33] is an extension of Java for information flow control. Labels are attached to variables when they are declared.Consumers voluntarily provide their personal information to a web site, and decide on restrictions on usage and recipients. Apart from the systemic limitations

of labeling and debugging the stronger limitations in its use are coming from the overly burdensome complexity of programming in Jif. Another work with similar motivation is the Resin [47] language runtime for PHP and Python, that supports policy objects, code that can be attached to objects, and propagates them along with the data. When the data reaches the system I/O boundaries, the attached policy object is evaluated automatically and it either verifies or rejects the flow.

HiStar [48] is an OS that also uses labels to provide information flow control for sensitive data. Apart from requiring a lot of effort to be applied to current systems, it can perform information flow control on sensitive data only if all processes are running on the same machine. DStar [49] overcomes this limitation by including a network protocol and framework that leverages OS-level protection on individual machines running Histar, to provide information flow control in distributed systems. DStar though, in contrast with CloudFence, can apply policies only at the more coarse-grained level of files and threads.

When focusing on the problem of data leakage for cloud-based services, most works reflect continuations of established lines of security research, such as web security and secure data outsourcing and assurance, rather than approaches with an exclusive focus on cloud security, with a few exceptions. Ristenpart et al. [35] investigated the security issues of existing deployed cloud systems, and identified a new class of vulnerabilities that can lead to cross-VM side-channel attacks.

Mundada et al. [27] presented Silverline, a system that allows cloud providers to offer data and network isolation for cloud-based services, with the goal to audit and prevent data leaks resulting from misconfigurations and side-channel attacks from co-resident cloud tenants. Although the concept of Silverline is close to CloudFence, Silverline supports information flow tracking using per-process labeling, requiring one process per user, which is not usually the case in most common web-applications. In contrast, CloudFence is based on fine-grained byte-level labeling and can handle multiple users per process.

Vanish [19] follows a different approach to information leakage prevention. It seeks to protect the privacy of past, archived data against accidental or malicious attacks by providing users with control over the lifetime of their private files. The idea is to ensure that all copies of sensitive data become unreadable after a user-specified time, without the need of any trusted third party for performing the deletion. Vanish meets this challenge by integrating cryptographic techniques with distributed systems.

Brown et al. [9] tried to address the problem of trustworthy cloud-hosted services even when the service provider is not trusted, by involving a trusted cloud provider attesting service application code to end-users. Like CloudFence, this work also tries to give insights to the end-users regarding the processing of their sensitive data by the cloud-hosted services, but the focus in this one is on code attestation and the main assumption is of a service provider as a PaaS client of the cloud, whereas CloudFence can be employed in all models of cloud services. Santos et al. [36] also worked on the issue of a trusted cloud computing platform (TCCP) but their approach was based on TPM attestation chains.

8 Conclusion

One of the most highly cited concerns regarding cloud-hosted third-party services is the fear of unauthorized exposure of users sensitive data. In lack of a better alternative option, the end users have to trust both the third-party service provider's as well as the cloud infrastructure provider's best efforts to properly handle their sensitive data as authorized.

This work takes a step further towards addressing this issue by introducing a new direct relationship between the users of the third-party online services and the cloud infrastructure provider. CloudFence is a service provided by the cloud infrastructure to both service providers and end-users, aiming to reinforce the users' confidence for their cloud-resident data.

Our evaluation using real-world applications shows that CloudFence can be integrated easily, even within existing applications, can protect against information disclosure attacks, and it imposes a modest performance overhead that allows its practical use in real environments. Our implementation is open source and freely available.

References

[1] `http://virtuemart.net`.

[2] `http://www.amanda.org`.

[3] `http://sitebar.org`.

[4] Virtuemart multiple sql injection vulnerabilities. `http://www.securityfocus.com/bid/37963`.

[5] AMES, S.R., J., GASSER, M., AND SCHELL, R. Security Kernel Design and Implementation: An Introduction. *Computer 16*, 7 (1983), 14 –22.

[6] ATTARIYAN, M., AND FLINN, J. Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. of OSDI* (2010), pp. 1–11.

[7] BOSMAN, E., SLOWINSKA, A., AND BOS, H. A Design for the World's Fastest Taint Tracker. In *Proc. of RAID* (2011).

[8] BOWERS, K. D., JUELS, A., AND OPREA, A. HAIL: a High-Availability and Integrity Layer for Cloud Storage. In *Proc. of CCS* (2009), pp. 187–198.

[9] BROWN, A., AND CHASE, J. Trusted Platform-as-a-Service: A Foundation for Trustworthy Cloud-Hosted Applications. In *Proc. of CCSW* (2011).

[10] CHEN, Y., PAXSON, V., AND KATZ, R. H. What's New About Cloud Computing Security? Tech. Rep. UCB/EECS-2010-5, EECS Department, University of California, Berkeley, Jan 2010.

[11] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *Proc. of USENIX Security Symposium* (2004), pp. 321–336.

[12] CLAUSE, J., LI, W., AND ORSO, A. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proc. of ISSTA* (2007), pp. 196–206.

[13] COMPUTERWORLD. Microsoft BPOS cloud service hit with data breach. `http://www.computerworld.com/s/article/9202078/Microsoft_BPOS_cloud_service_hit_with_data_breach`.

[14] CRANDALL, J. R., AND CHONG, F. T. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proc. of MICRO* (2004), pp. 221–232.

[15] DAVIS, B., AND CHEN, H. DBTaint: Cross-Application Information Flow Tracking via Databases. In *Proc. of WebApps* (Berkeley, CA, USA, 2010), WebApps'10, USENIX Association, pp. 12–12.

[16] DENNING, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM 19*, 5 (1976), 236–243.

[17] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of OSDI* (2010), pp. 393–407.

[18] FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., AND FELTEN, E. W. SPORC: Group Collaboration using Untrusted Cloud Resources. In *Proc. of OSDI* (2010), OSDI'10, pp. 1–.

[19] GEAMBASU, R., KOHNO, T., LEVY, A. A., AND LEVY, H. M. Vanish: Increasing data privacy with self-destructing data. In *Proc. of USENIX Security* (2009), pp. 299–316.

[20] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *Proc. of STOC* (2009), STOC '09, pp. 169–178.

[21] HO, A., FETTERMAN, M., CHRISTOPHER CLARK, A. W., AND HAND, S. Practical taint-based protection using demand emulation. In *Proc. of EuroSys* (2006), pp. 29–41.

[22] KANG, M. G., MCCAMANT, S., POOSANKAM, P., AND SONG, D. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. of NDSS* (February 2011).

[23] KEMERLIS, V., PAPPAS, V., PORTOKALIDIS, G., AND KEROMYTIS, A. iLeak: A Lightweight System for Detecting Inadvertent Information Leaks. In *Proc. of EC2ND* (Oct 2010), pp. 21–28.

[24] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. Tech. Rep. CUCS-044-11, Department of Computer Science, Columbia University, Oct 2011.

[25] KIM, H. C., KEROMYTIS, A. D., COVINGTON, M., AND SAHITA, R. Capturing information flow with concatenated dynamic taint analysis. In *Availability, Reliability and Security* (2009), pp. 355–362.

[26] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of PLDI* (2005), pp. 190–200.

[27] MUNDADA, Y., RAMACHANDRAN, A., AND FEAMSTER, N. SilverLine: Data and Network Isolation for Cloud Services. In *Proc. of HotCloud* (2011).

[28] MYERS, A. C. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. of POPL* (1999), pp. 228–241.

[29] MYSORE, S., MAZLOOM, B., AGRAWAL, B., AND SHERWOOD, T. Understanding and visualizing full systems with data flow tomography. In *Proc. of the* 13^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2008), pp. 211–221.

[30] NETHERCOTE, N., AND SEWARD, J. How to Shadow Every Byte of Memory Used by a Program. In *Proc. of VEE* (2007), pp. 65–74.

[31] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of NDSS* (2005).

[32] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. of EuroSys* (2006), pp. 15–27.

[33] PREIBUSCH, S. Information flow control for static enforcement of user-defined privacy policies. In *Proc. of POLICY* (2011), pp. 157–160.

[34] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proc. of MICRO* (2006), pp. 135–148.

[35] 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 *Proc. of CCS* (2009), pp. 199–212.

[36] SANTOS, N., GUMMADI, K. P., AND RODRIGUES, R. Towards trusted cloud computing. In *Proc. of HotCloud* (2009), HotCloud'09.

[37] SHADOWS OF EPIPHANY. Lighttpd – PHP Acceleration Benchmarks. `http://blog.bodhizazen.net/linux/lighttpd-php-acceleration-benchmarks/`.

[38] SILVA, J. D., AND GUOMUNDSSON, O. The amanda network backup manager. In *Proc. of LISA* (1993), pp. 171–182.

[39] SOPHOS. Groupon subsidiary leaks 300k logins, fixes fail, fails again. `http://nakedsecurity.sophos.com/2011/06/30/groupon-subsidiary-leaks-300k-logins-fixes-fail-fails-again/`.

[40] SUH, G. E., LEE, J., AND DEVADAS, S. Secure Program Execution via Dynamic Information Flow Tracking. In *Proc. of ASPLOS* (2004), pp. 85–96.

[41] TAKABI, H., JOSHI, J., AND AHN, G. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security and Privacy 8*, 6 (2010), 24–31.

[42] THE WALL STREET JOURNAL. Google Discloses Privacy Glitch. `http://blogs.wsj.com/digits/2009/03/08/1214/`.

[43] TIWARI, M., WASSEL, H. M., MAZLOOM, B., MYSORE, S., CHONG, F. T., AND SHERWOOD, T. Complete Information Flow Tracking from the Gates Up. In *Proc. of ASPLOS* (2009), pp. 109–120.

[44] WANG, W., LI, Z., OWENS, R., AND BHARGAVA, B. Secure and efficient access to outsourced data. In *Proc. of CCSW* (2009), pp. 55–66.

[45] XU, W., BHATKAR, S., AND SEKAR, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proc. of USENIX Security* (2006), pp. 121–136.

[46] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. of CCS* (2007), pp. 116–127.

[47] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Application Security with Data Flow Assertions. In *Proc. of SOSP* (2009), pp. 291–304.

[48] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making Information Flow Explicit in HiStar. In *Proc. of OSDI* (2006), pp. 19–19.

[49] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIÈRES, D. Securing Distributed Systems with Information Flow Control. In *Proc. of NSDI* (2008), pp. 293–308.

[50] ZHANG, Q., MCCULLOUGH, J., MA, J., SCHEAR, N., VRABLE, M., VAHDAT, A., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Neon: System Support for Derived Data Management. In *Proc. of VEE* (2010), pp. 63–74.

[51] ZHU, D., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *ACM Operating Systems Review 45*, 1 (2011), 142–154.