CloudFence: Data Flow Tracking as a Cloud Service

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

Computer Science Department, Columbia University
{vpappas,vpk,azavou,mikepo,angelos}@cs.columbia.edu

Abstract. The risk of unauthorized private data access is among the primary concerns for users of cloud-based services. For the common setting in which the infrastructure provider and the service provider are different, users have to trust their data to both parties, although they interact solely with the latter. In this paper we propose CloudFence, a framework for cloud hosting environments that provides *transparent*, *fine-grained* data tracking capabilities to both service providers, as well as their users. CloudFence allows users to *independently* audit the treatment of their data by third-party services, through the intervention of the infrastructure provider that hosts these services. CloudFence also enables service providers to confine the use of sensitive data in well-defined domains, offering additional protection against inadvertent information leakage and unauthorized access. The results of our evaluation demonstrate the ease of incorporating CloudFence on existing real-world applications, its effectiveness in preventing a wide range of security breaches, and its modest performance overhead on real settings.

Keywords: data auditing, data flow tracking, information confinement

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 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 [7, 15, 41, 42]. In lack of an alternative option (other than not using the service at all), most users eventually trust the service provider to keep their data safe.

Unfortunately, relying solely on reputable service providers does not mitigate the risk. Most feature-rich cloud-based services are quite complex, and are usually built by "glueing" together a multitude of components. Bugs and vulnerabilities in existing 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. At the same time, cloud computing encourages rapid application deployment, and time-to-market pressure sometimes makes data safety a secondary priority.

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, by giving 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 [12], 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, 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, as well as for protecting their own digital assets (e.g., configuration files or back-end databases). To facilitate the monitoring of user data, end users have access to a web-based dashboard [46] with meaningful log messages and a visual representation of the audit trails of their data.

A major challenge in supporting data auditing for services with a very large number of users is the need for concurrent propagation of tagged data that carry different tags for each user. At the same time, data tracking must be performed at a fine-grained level to allow for precise tracking of user data as small as a credit card number. CloudFence introduces a novel data flow tracking framework based on runtime binary instrumentation that supports *byte-level* data tagging, and *32-bit wide* tags per byte, enabling fine-grained data tracking for up to four billion users. Cross-application and cross-host tag propagation is handled transparently, without requiring any modifications to application code. Despite the significant increase in tag space, the runtime overhead of CloudFence is comparable to existing byte-level data flow tracking systems that support just a single [9,13,34] or up to eight [23,35,36] tags, and an order of magnitude lower compared to systems that support arbitrarily many tags [14,40].

We evaluate the performance and effectiveness of CloudFence using two real-world applications, and two publicly disclosed data leakage vulnerabilities in those applications. CloudFence can be easily integrated in both applications through the placement of just a few API calls, while it offers effective protection against a wide range of data theft threats, including SQL injection and arbitrary file read attacks.

Our work makes the following main contributions:

- We propose the use of data flow tracking as a service offered by cloud hosting providers i) for users, to independently audit their cloud-resident data, and ii) for service providers, to confine data propagation within well-defined domains.
- We present the design and implementation of a novel data flow tracking framework that uses 32-bit wide tags per byte, and introduces new features such as lazy tag propagation and persistent tagging on disk and across the network.
- We have implemented CloudFence, a prototype implementation of the proposed concept that allows service providers to easily integrate data flow tracking in their applications through a simple API.

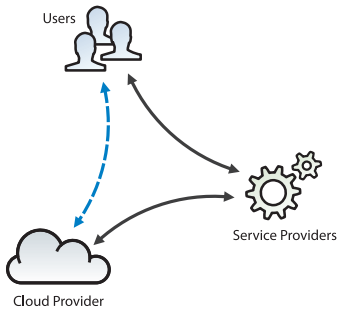

Fig. 1. Users explicitly trust their data to service providers, but also implicitly trust the cloud provider that hosts these services. CloudFence leverages this trust relationship to enable users to audit their data directly through the cloud provider.

– We have evaluated CloudFence using real applications and demonstrate its effectiveness and practicality.

2 Approach

Users of online services trust the providers of those services to securely handle and protect their data. Credit card numbers, private files, and other sensitive data is stored in back-end databases and file systems, beyond user control. In turn, service providers place their trust in the cloud infrastructure that hosts their services. The traditional provider-user relationship is thus transformed into a multi-party system [12], in which users are often not aware of the cloud infrastructure provider at all (unless it is the same entity that also offers the service, as for example is the case with many of the services 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. Although end users do not interact directly with cloud providers, they implicitly trust their infrastructure—the systems in which their data are kept. CloudFence aims to exploit this implicit trust for the benefit of all parties by introducing a *direct* relationship between end users and cloud providers, as shown in Figure 1. With data flow tracking as the basic underlying mechanism, the cloud provider enables users to directly inspect the audit trail of sensitive data that was handled by services hosted on the cloud provider's infrastructure.

Incentives While the trust relationship between users and service providers is not altered, CloudFence gives users an elevated degree of confidence by allowing them to independently monitor their private data as it propagates through the cloud. In fact, users are more likely to trust a large, well known, and highly reputable cloud provider, as opposed to a lesser-known developer or company (among the thousands that offer applications and services through online application distribution platforms).

CloudFence offers service providers two main benefits. First, with minimal effort, it allows them to provide an extra feature that reinforces the trust relationship with their users. This can also be considered as a competitive advantage: among two competing services, privacy-conscious users may prefer the CloudFence-enabled one, knowing that they will have an additional way of monitoring their data. Second, it empowers service providers with the ability 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 guarding user data, service providers can also take advantage of CloudFence to implement an additional level of protection for their own digital assets, such as back-end credentials, source code, or configuration files.

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

Security Model Our goal is to support benign service providers, who are willing to integrate CloudFence in their applications to enhance the security of the provided services. Note that this situation is typical for cloud-based services. End users have to implicitly trust their data to both the service provider and the cloud hosting provider in order to use these services. The current implementation of CloudFence is built on top of a user-level data flow tracking framework based on runtime binary instrumentation, which is directly integrated into the components of the protected service through an API provided by the cloud provider. In such a setting, application developers are responsible for specifying the sources of sensitive user input, so that all necessary data is always being marked and tracked appropriately.

Our approach offers protection against many classes of attacks that can lead to unauthorized data access (but which do not allow arbitrary code execution), such as SQL injection, command substitution, parameter tampering, directory traversal, and other prevalent web attacks that are seen in the wild. In case of attackers who gain arbitrary code execution, we can no longer guarantee accurate data tracking, since they can not only compromise our framework, but can also exfiltrate data through covert channels. Finally, besides protecting against external attacks, an equally important goal of CloudFence is to bring into users' and service providers' attention any unintended data exposure that may lead to unauthorized access. For example, sensitive data can accidentally be recorded in error logs or included into memory dumps after an application crash.

System Overview Figure 2 shows the main interactions among the different parties that are involved in CloudFence-enabled services. Initially, users register with the cloud provider (1) and acquire a universally unique ID, distinctive within the vicinity of the cloud provider's infrastructure. Then, they use the online services offered by various service providers by providing the ID acquired from the previous step (2).

The actual mechanism used for conveying user IDs to CloudFence is not addressed in this work. As possible solutions, the service provider can either request from users to provide their ID during the sign up process on the corresponding application, or in

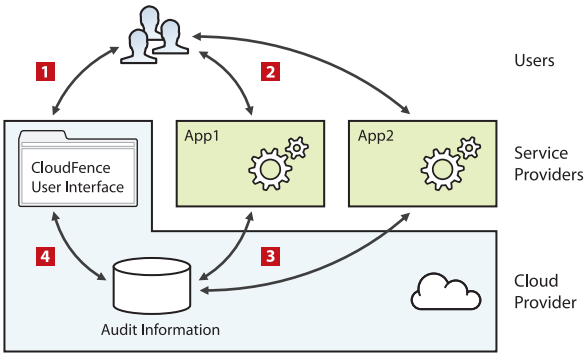

Fig. 2. Main interactions between the different parties involved in CloudFence-enabled applications. Users register with the cloud provider (1), and then use the services offered by various service providers using the same set of credentials (2). Sensitive data are tagged and tracked transparently throughout the cloud infrastructure (3). Users can audit their data through a web interface exposed directly by the cloud provider (4).

case a cloud-wide identity management system is in place, the application can access the respective ID transparently by requesting it directly from the cloud provider (after the user has successfully authenticated). Such functionality is gaining traction among cloud providers. Indicatively, Amazon recently launched the "Login with Amazon" feature [2], which allows users to login to Amazon-hosted services with a single account, while it also supports federated login using Google and Facebook identities [1].

Sensitive data is tagged by the service provider with the supplied user ID, 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 data directly through the cloud provider using a user-friendly web interface (4). Service providers also have read access to the collected audit data through a specialized API. Besides user data, CloudFence can be used to protect the service providers' own assets, such as back-end credentials, configuration files, and source code. This can be achieved by tagging them as sensitive, tracking their propagation through the cloud infrastructure, and enforcing fine-grained perimetric access control based on the applied tags.

Challenges The on-demand consolidation of computing elements in cloud settings allows service providers to easily "glue" together functionality and content from third-party sources, and build feature-rich applications. As the benefits of this approach are numerous, it is critical not to interfere with that paradigm while enabling data tracking. We consider this as the *transparent tracking* requirement. The applied DFT method should support incremental deployment by not requiring intrusive changes, such as manually annotating source code [33], custom OSs [48], or modified hypervisors [50].

Second, tracking granularity plays a crucial role in the effectiveness of DFT. A service provider can choose between tracking data as small as a single byte [30], which enables robust protection against extreme cases of data leakage, or employing a more coarse-grained (and hence error-prone) approach [29]. However, fine-grained DFT has a significant performance cost, as tracking logic becomes more intricate (e.g., consider

the case of two 32-bit numbers with 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.

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 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.

Finally, the main concept behind CloudFence requires that personal data are marked with a respective user ID. The goal is to support applications with a *practically unlimited number of users*, and thus the DFT component should be able to handle a respectively large number of tags. This requirement is highly challenging, as most DFT frameworks provide either a single tag [9, 13, 34] or just a few—usually eight [23, 35, 36].

3 Design

CloudFence consists of three main components: the *data flow tracking* (DFT) subsystem, the *API stub*, and the *audit trails generation component*. The DFT subsystem performs fine-grained, byte-level explicit data flow tracking without requiring any modification to applications or the underlying OS, while at the same time handles 2^{32} different tags. Our DFT component supports tracking across processes running on the same or remote hosts. Specifically, it *piggybacks* tags on the data exchanged through IPC mechanisms or network I/O channels, keeps persistent tag information for marked data written to files, and handles (un)marshalling transparently. Finally, the low ratio of tagged data allows for further optimizations, like lazily propagating the tags when possible.

The API stub allows service providers to *tag*, i.e., attach metadata information, on sensitive user data that enters their applications. CloudFence does not require application modifications regarding data tracking (e.g., extensive annotations). However, it requires small changes to application code for marking sensitive information. Figure 3 illustrates the overall 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. Note that for the rest of our discussion, we assume that the service provider relies on an IaaS delivery mechanism, and in this example both processes run on the same (virtual) host. However, CloudFence can be seamlessly employed in PaaS and SaaS setups. Data that are tagged as sensitive (denoted by the solid line in the figure) is tracked across all local files, host-wide IPC mechanisms, and selected network sockets. Tagged bytes that are written to storage devices, or transmitted to remote hosts, result in an audit message.

Data Flow Tracking Although our DFT component is inspired by previously proposed DFT tools [9, 23], for reasons that are explained in detail in Section 4, we built it from scratch to provide a transparent, fine-grained, and domain-wide tracking framework suitable for the target cloud environment. We employ Intel's Pin [28], a dynamic binary instrumentation toolkit. Pin injects a tiny user-level virtual machine monitor (VMM) in

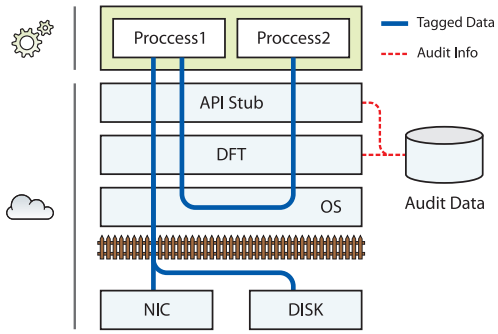

Fig. 3. CloudFence architecture.

the address space of a running process, or in a program that launches itself, and provides an extensive API that CloudFence uses for inspecting and modifying (dynamically at run-time) the process' code at the instruction level.

In particular, CloudFence uses Pin to analyze all instructions that move or combine data to determine data dependencies. Then, based on the discovered dependencies, it instruments program code by injecting the respective tag propagation logic *before* the corresponding instructions. Both the original and the additional instrumentation code, i.e., the data tracking logic, are re-translated using Pin's just-in-time compiler. 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.

API The CloudFence API consists of three functions (C prototypes): `add_tag()`, `del_tag()` and `copy_tag()`. The `add_tag` function is used for associating a 32-bit tag to every byte while `del_tag` is used for unlabeling data. The `copy_tag` function propagates the tag information 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*, which we further discuss in Section 6. CloudFence also provides appropriate wrappers for higher level languages, which are commonly used in web applications. In particular, for some of the applications used in our evaluation, we developed a PHP extension that provides data tagging to string arguments (other types can be supported likewise), by internally calling the lower-level C functions exported by the CloudFence API.

Audit Trails Generation The main purpose of CloudFence's auditing mechanism is to generate *detailed audit trails* for tagged data. Therefore, we implemented a generic "verbose" logging mechanism that collects information for tagged data accesses and generates audit logs. The generated trails are stored in a database outside the vicinity of the service provider in an "append-only" fashion to prevent tampering of archived audit trails. The DFT component pushes audit information to the audit component whenever tagged data is written to a cloud storage device or pass through I/O channels to endpoints inside or outside the cloud.

4 Implementation

From a high-level perspective, most of CloudFence's functionality is built around the DFT component, except the user interface, which is a user-accessible web application coupled with a back-end database. Our current prototype is implemented using Pin 2.10, and works with unmodified applications running on x86-64 Linux. The data auditing component is layered on top of CloudFence using system call interposition.

4.1 32-bit Wide Tags and 64-bit Support

The *shadow memory* used for keeping data tag information plays a crucial role in runtime performance. Previously proposed DFT systems mainly use two approaches for tagging memory: (i) bit-sized tags [34], whereby every byte of addressable memory is tagged using a single bit in the shadow memory, and (ii) byte-sized tags [9, 13, 23], whereby each byte of program memory has a sibling in the shadow arena. In between, systems like Umbra [36] and TEMU [40] allow for various byte-to-byte and byte-to-bit configurations, as well as for lossy encodings (e.g., four bytes of addressable memory can be tagged using one byte). TEMU, in particular, enables very flexible tagging, by supporting tag values of arbitrary size, at the expense of higher runtime performance overhead [44]. CloudFence trades some of this flexibility for a lower runtime slowdown.

Implementing 32-bit wide tags requires re-designing the shadow memory from scratch. Driven by the fact that data from different sources, carrying dissimilar tags, are rarely combined in our context (e.g., the memory bytes of two different credit card numbers are unlikely to be combined), we opted for a solution that greatly increases the number of tags stored per datum, but unavoidably also increases the overhead of tag combination operations. More precisely, each tag value is stored as a different number, and when two tags are combined, a *new* tag value is created. Still, incorporating this change alone in commodity DFT systems [9, 23] would only increase the number of tags from 8 to 256, using byte-size tagging. Hence, our next step was to expand the tag size from one to four bytes, allowing for 2^{32} tags.

The transition to 64-bit not only helps overcoming available memory limits, but also enables further optimizations. The relatively expensive translation that involves shadow page table lookups is replaced by a faster one. Taking advantage of the ample address space, we split it in two parts: the shadow memory and the actual process memory. This is achieved by reserving the shadow memory as soon as the process is started, forcing it to allocate memory only in its own part. Address translation then becomes as simple as scaling the virtual address and adding an offset. For example, the memory tags of address `vaddr` can be obtained as follows: `taddr = (vaddr << 2) + toff`, where `toff` corresponds to the offset of the shadow memory. CloudFence reserves 16TB of user space for the application and 64TB for the shadow memory, resulting in an offset value of `0x100000000000`. However, it allocates pages in the shadow region on demand, i.e., only when a page contains tag information. As every byte of tracked program data needs four more bytes for its tag, part of the physical memory footprint of a process increases by a factor of four.

4.2 Lazy Tag Propagation

Most x86-64 instructions fall into one of two major categories: arithmetic and data transfer. For the latter, tags are always propagated following the flow of data, i.e., we *always* copy the tags of the source operand over the tags of the destination operand. On the other hand, whenever the destination operand is derived from a combination of its own value and that of the source operand, there are three possible cases, each having a different impact in terms of performance:

```
/* arithmetic instructions */
if (shadow[src] != 0)
  if (shadow[dst] == 0)
    shadow[dst] = shadow[src];
  else if (shadow[dst] != shadow[src])
    shadow[dst] = combine(shadow[src], shadow[dst]);
```

Starting from the worst case, (`else if`), if both operands have different tags, a lookup is performed and a *new* tag is generated. If only the source operand is tagged, its tag is copied to the destination. If the source operand is not tagged, no action needs to be taken. Given that only a small amount of data is usually tagged in our scenarios (recall that we care for discrete pieces of sensitive information), we optimized our design for the last case using Pin's API for fast conditional instrumentation. Arithmetic instructions are instrumented with a lightweight check of whether the source operand is tagged (*fast path*). In case it is, the appropriate propagation actions are performed according to the code snippet above (*slow path*). This avoids in the common case the excessive register spilling that usually occurs by larger instrumentation code that needs more registers [28]. Finally, tag information is kept into an array-like data structure, indexed by tag value. For every tag, we store whether it is basic or compound, and in the latter case, the tag values it stems from. Compound tags can be traced back to the basic tags they are made of, by recursively querying this data structure.

4.3 Tag Persistence

Accurate data flow tracking throughout a cloud-based application requires persistent data tags and tag propagation across different processes, which may run on different (physical or virtual) hosts. To this end, we have built a layer on top of our prototype for supporting tag propagation across BSD sockets, Unix pipes, files, and shared memory.

Sockets and 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 prototype, the exchanged tag information consists of a copy of the relevant area of the shadow memory that CloudFence maintains for the transmitted data, encoded in RLE (Run Length Encoding). Recall that only a very small part of data is usually tagged, so most of the time there will be minimal communication overhead—just a header field that contains the number of triplets.

Synchronous I/O. We hook the `write`, `send`, and `writev` system calls using Pin's hooking API, and transmit tag information before the actual data of the original system call. Similarly, we hook the `read`, `recv`, and `readv` system calls, and read

the tag information before the actual data. Messages 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 within the same receiving operation, so they are simply decoded and attached to the original data. For messages received through several read operations, the receiver initially buffers the tag information, and each time a new part is received, its corresponding tag information is appended until the whole message is received. The most difficult case is when the size of the send buffer does not match the size of the receive buffer. Such cases are handled by changing the return value of the read operation to match the end of the current message.

Non-blocking I/O. For non-blocking I/O, the above system calls may return a special error code as if the requested operation would block (`EAGAIN`). 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 encoded shadow memory data that is actually sent, and emulate `EAGAIN` errors until all relevant shadow data has been completely transmitted.

Multiplexed I/O. For `select`, `poll`, and `epoll`, we chose to trade a small performance overhead in favor of a safer hooking implementation. Before read or write operations, the system blocks until all tag information is received or sent, as in synchronous I/O. A more robust implementation would 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. If only partial information is available, we can buffer it, and remove the file descriptor from the returned set of `select` or `poll`, as if it were 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.

Files. Tag information should persist even when data is written into files, so that it can be later retrieved by the same or other processes. 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, CloudFence creates a second shadow file, which is mapped to memory and is associated with the original file descriptor. Whenever a process writes a file using `write`, `writev`, or `pwrite`, 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 using `read`, `readv`, or `pread`, the relevant tag information from the corresponding shadow file is represented at the destination buffer. To limit space requirements, we take advantage of sparse files, which are supported by most modern OSs. For the common case of a file with just a few tagged bytes, the shadow file will consume just 4× the size of only the tagged data, while shadow files that contain no tag information require no extra space at all.

Shared Memory. Our current implementation supports shared memory regions allocated with `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 tag information.

4.4 Data Flow Domain

Data flow tracking is performed within the boundaries of a well-defined *data flow domain*, according to the components of the online service. 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.

To automate the configuration of tag propagation between processes that exchange data through the network, CloudFence maintains a global registry of active sockets for the domain by hooking the `connect` and `accept` system calls. For each connection attempt, the initiator's IP address and port are 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 service providers must only specify the programs that comprise the cloud application, and then the rest of the tag propagation logic is determined automatically.

4.5 User Interface

CloudFence's user interface leverages Cloudopsy [46], a web-based data auditing dashboard. Cloudopsy uses visualization and automated audit log analysis to provide users who lack technical background with a more comprehensible view of audit information. For example, the event of a user's credit card number being sent to an external host other than those included in the trusted domain, which could be a data leak incident, would be clearly depicted in the audit graph presented to the user. In particular, this suspicious data flow would be presented in the audit graph by a directed link in a pre-defined color (e.g., red) indicating the possible data leak. Details regarding the different formats of the audit graphs presented to the end users and the service providers are out of the scope of this paper but are discussed in our paper [46]. Although this service targets mostly end users, it also provides administrators with a graphical overlook of the overall application dependencies and data flows of the service. The visualization of audit events allows for the immediate verification of legitimate operations and the identification of unexpected transmissions, which otherwise might have remained hidden much longer in the reams of raw audit logs, thus reducing decision and reaction latency.

5 Evaluation

We evaluate CloudFence in terms of ease of deployment in existing applications, runtime performance, and effectiveness against data leakage attacks, using two real-world applications: an e-commerce framework 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 run

VMWare ESXi v4.1, and all CloudFence-enabled applications were installed in virtual machines. The third server was used to simulate clients and drive the experiments. In all cases, the operating system was 64-bit Debian 6.

5.1 Deploying CloudFence

Online Store The first scenario we consider is an online store hosted on a cloud-based infrastructure. Typically, during a purchase transaction, sensitive information, such as the credit card number and the recipient's postal and email address, is 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 data, as well as to restrict the use of sensitive data within the application's domain. The developers of the e-store know in advance the entry points of sensitive user data, as well as which processes and hosts should be allowed to access 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 a connection to the third-party payment processor, i.e., a well-known remote server address.

The application we chose for this scenario, called VirtueMart, is an open source e-commerce framework developed as a Joomla component, and is typically used in PHP/MySQL environments. We configured VirtueMart to accept payments only through credit card, and set up actual 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 at the user registration and order checkout modules. Specifically, we added a new input field in the registration form for the user's unique ID, a new column in the user's database table, and a few lines of code for storing the ID in the database along with the user's info. For the checkout phase, we added a few lines of code in the script that processes the payment information. First, the user ID for the current session is queried from the database. Then, the HTTP POST variable that holds the credit card number is tagged by calling the `add_tag` API function through a PHP wrapper. Finally, the data flow domain of the application comprises the web server, the database server, and any other processes these two may spawn.

Bookmark Synchronization This use case stems from the increased demand for data synchronization services, as users typically have many internet-connected devices. The scenario in this case is to host a bookmark synchronization service on the cloud based on SiteBar, an online bookmark manager written in PHP. When adding a link to SiteBar, users have the option to set it as public or private, and may change it later. From the side of the service provider, we would like to tag any private links as sensitive.

Incorporating CloudFence in SiteBar was very similar to the previous case, as both applications are written in PHP and use MySQL as a database back-end. On the other hand, changing the source code to tag the sensitive data (user links marked as private) was slightly more elaborate, as the sensitivity level of data can change dynamically. Thus, we had to change the code that adds a link so as to tag it in case it is marked as private, as well as the code for editing a link. 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 can identify illegitimate data accesses performed as a result of attacks. We used exploits against two publicly disclosed vulnerabilities in the studied applications. The first allows authenticated users of SiteBar versions earlier than v3.3.8 to read arbitrary files [3]. 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 below:

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

Passing a file name that ends with the URL-encoded value for the zero byte (`%00`) causes the `open` system call to ignore any characters after it and read the supplied file:

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

Using SiteBar v3.3.8 on top of PHP v5.2.3, we repeatedly read files by exploiting this bug through a browser on a remote machine. CloudFence reported successfully all accesses to data with persistent tags in the file system, which in our case corresponded to files belonging to MySQL.

Another type of attack that usually leads to information leakage is SQL injection. The main cause, again, is the insufficient user input validation. To demonstrate the effectiveness of CloudFence on preventing this type of attacks, we used another real-world vulnerability in VirtueMart version 1.1.4 [4]. The value of the HTTP GET parameter `order_status_id` is not properly sanitized, allowing malicious users to change the SQL SELECT query by using a URL like the following:

```
http://VM_APP/index.php?option=com_virtuemart&page=order.order_status_form
&order_status_id=-1' UNION ALL SELECT ... 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 above query returns a row from the `jos_vm_order_payment` table, which holds the credit card numbers, instead of the table `jos_vm_order`. As in the previous case, we installed the vulnerable version of VirtueMart on top of PHP v5.3.3, and tried to access the credit card numbers by exploiting this bug. In all cases, CloudFence identified the exfiltration attempt, as the relevant data had been tagged as sensitive upon entry.

5.3 Performance

To assess the runtime overhead of CloudFence we compare it against Libdft [23], a data flow tracking framework for commodity systems, as well as the unmodified application in each case. We chose Libdft because it is publicly available, and it also uses Pin for runtime binary instrumentation. Libdft maintains a shadow byte for each byte of data, and thus supports only eight tags per byte, represented by individual bits. Compared to CloudFence, which uses four shadow bytes per actual byte of data, Libdft has thus significantly lower shadow memory requirements. Furthermore, representing each tag

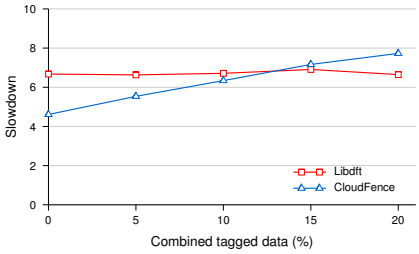

Fig. 4. Slowdown as a function of the percentage of data with different tags that must be combined (worst case). CloudFence not only supports 2^{32} tags (instead of just eight for Libdft), but also is faster for the cases we consider in our setting ($< 10\%$).

using a single bit allows Libdft to implement aggressive optimizations for tag propagation using bitwise OR operations. In contrast, CloudFence has to synthesize a new tag whenever two existing tags must be combined, and then maintain their association. As we show in this section, despite the increased requirements of CloudFence in terms of memory consumption and computation for supporting 32-bit tag propagation, its runtime overhead is comparable to Libdft for the cloud-based applications we consider.

Microbenchmark We begin by focusing on the overhead of tag propagation, and specifically exploring tag *generation*, which is the worst case scenario for CloudFence. The test program we used allocates two buffers, `buf_a` and `buf_b`, of the same size. The bytes of `buf_a` are tagged with the value `1`. Each byte of a specified part of `buf_b` is tagged with a different value, starting from `2`. Then, each byte of `buf_a` is added to the corresponding byte in `buf_b`, and the process repeats for a number of times. For each add operation, if the current byte in `buf_b` is not tagged, then `buf_a`'s tag is copied over, otherwise, their tags are combined and a new one is generated.

Figure 4 shows the slowdown imposed by data flow tracking for CloudFence and Libdft. CloudFence not only provides extra functionality that is crucial for cloud environments, but at the same time it is even faster than Libdft for the cases we consider, i.e., minimal combination of data marked with different tags, as the personal data of different users are not likely to be intermixed. The extreme case in which each add operation generates a new tag results in a 20× slowdown (upper bound).

Real-world Applications We decided to focus our experiments on VirtueMart, as it represents the most complicated scenario among the chosen applications. VirtueMart stresses a larger part of CloudFence's functionality, and therefore results in a larger but more representative performance impact in comparison to SiteBar. In our experiment, we measure the sustained throughput of user requests that VirtueMart can handle when processing concurrent purchase transactions from multiple users. We installed two instances of VirtueMart on virtual machines in our testbed. One runs on top of Apache using the PHP module, and the other was compiled after transforming the PHP to C++ using Facebook's HipHop. In both cases, MySQL was the database back-end. To gen-

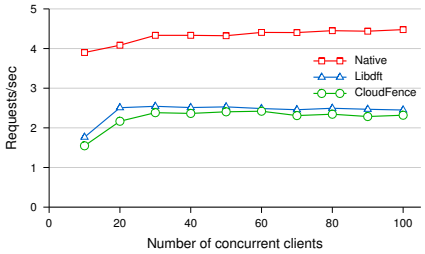

Fig. 5. Request throughput for VirtueMart using the default web server configuration.

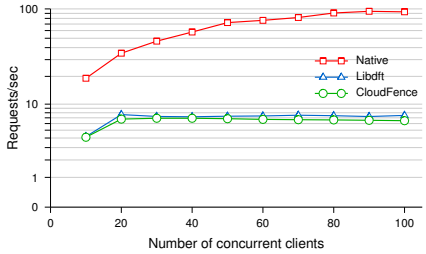

Fig. 6. Request throughput for VirtueMart using Facebook's HipHop.

erate 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 imposed stress on the server when emulating multiple concurrent user transactions.

Instead of performing the same request over and over, we simulated 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. For 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.

Figure 5 shows the sustained request throughput for a varying number of concurrent web clients, when VirtueMart is running i) natively, ii) on top of Libdft, and iii) on top of CloudFence. The request throughput was calculated by dividing the number of requests by 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. We see that although CloudFence reduces the throughput in half, its performance is comparable to Libdft despite its much more CPU and memory intensive tag propagation logic. A significant fraction of the slowdown for both systems can be attributed to Pin's overhead for runtime binary instrumentation. We should note that the server throughput in the native case is not bounded due to limited computational resources, 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 additional overhead.

Figure 6 shows the results of the same experiment using the compiled version of VirtueMart and the built-in multi-threaded web server that comes with the HipHop code transformer. This time, the native throughput is bound due to CPU saturation. In the worst case, the request throughput is roughly 13 times slower when CloudFence is enabled. Another contributing factor to performance degradation as concurrency increases lies in the underlining binary instrumentation framework. To provide thread-safe exe-

cution of system call hooks, Pin serializes their execution using a process-level global lock. This kind of hooks are used by both CloudFence and Libdft, which again achieve comparable performance.

6 Discussion

Over-tagging. We opted for a design that does not suffer from over-tagging or tag pollution. Specifically, CloudFence does not tag pointers nor it propagates tags due to implicit flow, which prior work has shown to produce over-tagging [14, 39]. Moreover, it takes into consideration that certain system calls write specific data to user-provided buffers. For instance, consider `gettimeofday`, which upon every call overwrites the user space memory that corresponds to one, or two, `struct timeval` data structures. Such system calls always result in sanitizing (untagging) the data being returned, unless CloudFence has installed a callback that selectively tags returned data.

Under-tagging. CloudFence only supports explicit data flows, which can lead to under-tagging whenever the service provider uses a code construct that copies sensitive data using branch statements. As an example, consider the code snippet `if (in == 1) out = 1;`. Although the value of `in` is copied to `out`, any tags associated with it are not. DTA++ [22] addresses this issue by identifying implicit flows within information-preserving transformations and generating rules to add additional tags only for a certain subset of control-flow dependencies. During our evaluation, we identified a couple of such cases, in AES encryption (used in SSL, MySQL, and the Suhosin PHP hardening extension) and Base64 encoding. Such cases should be handled manually by the service provider, by hooking the corresponding functions and copying the tag information from their source to the target operand using the `copy_tag` function.

Binary Instrumentation. The choice of a DFT framework based on binary instrumentation unavoidably comes with an increased runtime penalty. However, we have managed to support 32-bit wide tags per byte while maintaining a similar, or even lower, overhead compared to existing systems, allowing the practical use of CloudFence in real settings. Alternative implementations of this functionality within language runtimes [6, 8], or even at hardware, have been shown to degrade the imposed overhead.

Fine-grained Tracking. CloudFence is a general framework designed for use with all the components of a cloud-based service without modifications. To achieve this, we chose fine-grained over coarse-grained (process-level) tags, although this comes with an increased overhead. Other implementations [29] that tried to avoid this overhead by coloring each time the entire process serving the HTTP request for user data with the tag or color representing this specific user, ended keeping extra information in the application level, when its processes where handling data from multiple users at the same time. As expected, in this case the process would be assigned a tag representing both users, but if there was no merge of the data, this data would still carry the new tag instead of the initial unique user tag. Therefore, the audit capability provided to the end users for their cloud-based data would not be as precise in the case of process-level tags as it is in our fine-grained implementation.

Alternative DFT Tools. CloudFence has been influenced by previous DFT proposals, with the closest being Libdft [23], but none of them would suffice for our goal.

In particular, although CloudFence and Libdft share the same underlying DBI framework (Pin), they differ completely in (i) shadow memory design, (ii) tag propagation logic, and (iii) I/O interface. Libdft uses dynamically allocated shadow memory (tracks memory allocations) and a page-table-like structure for performing virtual-to-shadow memory translations. CloudFence, on the other hand, reserves part of the abundant 64-bit address space for storing the 4-byte wide tags (per byte of program memory), thus making memory-to-tag translation without a lookup. Regarding the low-level optimization that Libdft uses, we retained what it considers as `fast_vcpu` and `huge_tlb`. Finally, the system call interface of 64-bit Linux is slightly different from the 32-bit version and the system call numbers are shuffled. Hence, the I/O system call descriptors that CloudFence uses had to be adapted.

Universally-unique User IDs. The use of the same ID across all services may raise privacy concerns, as this allows the cloud provider to track user activity within its premises. Although cloud providers could track users even if a cloud-wide user ID was not used, e.g., by combining user-identifying features such as browser fingerprints and HTTP cookies [25], a unique ID per user certainly makes tracking easier. Cloud providers, however, have already started offering access to hosted services through in-house [2] or third-party web identity providers [1], and this trend is expected to continue, as it improves user experience by having to manage fewer accounts.

7 Related Work

A common approach for degrading the impact of data leaks is to store important data in an encrypted form on the remote server [10, 19, 43]. Even though encryption alleviates the problem of secure storage in the cloud, it does not solve the issue when also processing on this data on the remote infrastructure is required. The homomorphic encryption scheme [21], although promising it is for now computationally prohibitive for real-world applications.

Information flow tracking (IFT) is another common approach for protection against information leakage. IFT implementations range from per-process [14, 31, 34, 51] and singe-host tracking [16, 32, 44] to the more recent cross-host taint tracking systems [5, 17, 18, 24, 47, 50]. These designs were well suited for the contexts in which they were proposed, but in contrast to our approach, they are difficult to adapt in different environments. Jif [33] and Resin [45] extend the Java and PHP language runtimes, respectively, with IFT abilities to enable user privacy constraints and prevent information leakage. Although they allow better performance numbers for the DFT component, they require complete application rewrites and suffer from the inherent limitation of labeling and tracking at the coarse-level of objects, in contrast to our more fine-grained and application agnostic approach. DStar [49] and Flume [27] are alternative IFT mechanisms for distributed systems, which though do not meet our needs since they cannot track granularities smaller than high-level objects, i.e., files, processes and sockets, or they would require rewriting of the monitored applications to enable the tracking mechanism. Vanish [20] follows a different approach to information leakage prevention, by ensuring 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.

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 [26, 29, 37]. Among them is Silverline [29], a system close to our vision, with the goal of enabling cloud providers with auditing and data leaks prevention capabilities. Although we share the same goal, the process-level tainting they support, is rather coarse-grained for the most common web-applications, and as such it is not applicable to a wide-range of cloud applications. Similar in spirit to our work, the W5 project [26] although it introduces some of the concepts used in CloudFence, we offer a working implementation which supports a more fine-grained labeling and data tracking approach, able to handle multiple users per process — as in most common web-applications.

Brown et al. [11] 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 is on code attestation and the service provider is a PaaS client of the cloud, whereas CloudFence can be employed in all models of cloud services. Finally, Santos et al. [38] also worked on the issue of a trusted cloud computing platform (TCCP) but their approach relies on TPM attestation chains.

8 Conclusion

One of the most highly cited concerns regarding cloud-hosted services is the fear of unauthorized exposure of sensitive user data. Users have to trust the efforts of both the third-party service provider and the cloud infrastructure provider for properly handling their private data as intended. In this work, we take a step towards increasing the confidence of users for the safety of their cloud-resident data by introducing a new direct relationship between end users and the cloud infrastructure provider. CloudFence is a service provided by the cloud infrastructure, that offers data flow tracking abilities to both service providers and their users for user data collected in the realm of cloud-based services. In particular, CloudFence allows users to independently audit their data by the cloud-based services, and additionally enables service providers to confine data propagation and protect their digital assets within well-defined domains. Our evaluation using real-world applications demonstrates that CloudFence can be integrated easily in existing applications, can protect against information disclosure attacks, and imposes a modest performance overhead that allows its practical use in real environments. Our prototype implementation is open source and freely available.

Acknowledgements. This work was supported by DARPA and the National Science Foundation through Contract FA8651-11-C-7190 and Grant CNS-12-28748, respectively, with additional support from Intel and Google. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, NSF, Intel, or Google.

References

1. AWS taps social networks for identity verification, http://www.theregister.co.uk/2013/05/29/aws_social_identity_verification
2. Login with Amazon, http://login.amazon.com/
3. SiteBar: Multiple issues, http://www.securityfocus.com/archive/1/483364
4. VirtueMart Multiple SQL Injection Vulnerabilities, http://www.securityfocus.com/bid/37963
5. Attariyan, M., Flinn, J.: Automating configuration troubleshooting with dynamic information flow analysis. In: Proc. of OSDI (2010)
6. Bello, L., Russo, A.: Towards a Taint Mode for Cloud Computing Web Applications. In: Proc. of PLAS. pp. 1–12 (2012)
7. Berghel, H.: Identity Theft and Financial Fraud: Some Strangeness in the Proportions. Computer 45(1), 86–89 (Jan 2012)
8. Bisht, P., Hinrichs, T., Skrupsky, N., Venkatakrishnan, V.N.: WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In: Proc. of CCS. pp. 575–586 (2011)
9. Bosman, E., Slowinska, A., Bos, H.: Minemu: The World's Fastest Taint Tracker. In: Proc. of RAID. pp. 1–20 (2011)
10. Bowers, K.D., Juels, A., Oprea, A.: HAIL: a High-Availability and Integrity Layer for Cloud Storage. In: Proc. of CCS. pp. 187–198 (2009)
11. Brown, A., Chase, J.: Trusted Platform-as-a-Service: A Foundation for Trustworthy Cloud-Hosted Applications. In: Proc. of CCSW. pp. 15–20 (2011)
12. Chen, Y., Paxson, V., Katz, R.H.: What's New About Cloud Computing Security? Tech. Rep. UCB/EECS-2010-5, EECS Department, University of California, Berkeley (Jan 2010), http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-5.html
13. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S.: TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In: Proc. of ISCC. pp. 749–754 (2006)
14. Clause, J., Li, W., Orso, A.: Dytan: A Generic Dynamic Taint Analysis Framework. In: Proc. of ISSTA. pp. 196–206 (2007)
15. Computerworld: Microsoft BPOS cloud service hit with data breach (Dec 2010), http://www.computerworld.com/s/article/9202078/Microsoft_BPOS_cloud_service_hit_with_data_breach
16. Crandall, J.R., Chong, F.T.: Minos: Control Data Attack Prevention Orthogonal to Memory Model. In: Proc. of MICRO. pp. 221–232 (2004)
17. Davis, B., Chen, H.: DBTaint: Cross-Application Information Flow Tracking via Databases. In: Proc. of WebApps (2010)
18. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proc. of OSDI (2010)
19. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: SPORC: Group Collaboration using Untrusted Cloud Resources. In: Proc. of OSDI (2010)
20. Geambasu, R., Kohno, T., Levy, A.A., Levy, H.M.: Vanish: Increasing Data Privacy with Self-Destructing Data. In: Proc. of USENIX Sec. pp. 299–316 (2009)
21. Gentry, C.: Fully Homomorphic Encryption Using Ideal Lattices. In: Proc. of STOC. pp. 169–178 (2009)
22. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In: Proc. of NDSS (2011)
23. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In: Proc. of VEE (2012)
24. Kim, H.C., Keromytis, A.D., Covington, M., Sahita, R.: Capturing Information Flow with Concatenated Dynamic Taint Analysis. In: Proc. of ARES. pp. 355–362 (2009)
25. Kontaxis, G., Polychronakis, M., Keromytis, A.D., Markatos, E.P.: Privacy-preserving social plugins. In: Proceedings of the 21st USENIX Security Symposium (August 2012)
26. Krohn, M., Yip, A., Brodsky, M., Morris, R., Walfish, M.: A World Wide Web Without Walls. In: Proc. of HotNets (2007)
27. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Frans, M., Eddie, K., Morris, K.R.: Information Flow Control for Standard OS Abstractions. In: Proc. of SOSP. pp. 321–334 (2007)

28. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proc. of PLDI. pp. 190–200 (2005)
29. Mundada, Y., Ramachandran, A., Feamster, N.: SilverLine: Data and Network Isolation for Cloud Services. In: Proc. of HotCloud (2011)
30. Nethercote, N., Seward, J.: How to Shadow Every Byte of Memory Used by a Program. In: Proc. of VEE. pp. 65–74 (2007)
31. Newsome, J., 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., Bos, H.: Argos: an Emulator for Fingerprinting Zero-Day Attacks. In: Proc. of EuroSys. pp. 15–27 (2006)
33. Preibusch, S.: Information Flow Control for Static Enforcement of User-Defined Privacy Policies. In: Proc. of POLICY. pp. 157–160 (2011)
34. Qin, F., Wang, C., Li, Z., Kim, H.s., Zhou, Y., Wu, Y.: LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In: Proc. of MICRO. pp. 135–148 (2006)
35. Qin Zhao and Derek Bruening and Saman Amarasinghe: Efficient Memory Shadowing for 64-bit Architectures. In: Proc. of ISMM. pp. 93–102 (2010)
36. Qin Zhao and Derek Bruening and Saman Amarasinghe: Umbra: Efficient and Scalable Memory Shadowing. In: Proc. of CGO. pp. 22–31 (2010)
37. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In: Proc. of CCS. pp. 199–212 (2009)
38. Santos, N., Gummadi, K.P., Rodrigues, R.: Towards Trusted Cloud Computing. In: Proc. of HotCloud (2009)
39. Slowinska, A., Bos, H.: Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In: Proc. of EuroSys. pp. 61–74 (2008)
40. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A New Approach to Computer Security via Binary Analysis. In: Proc. of ICISS. pp. 1–25 (2008)
41. Sophos: Groupon subsidiary leaks 300k logins, fixes fail, fails again (2011 Jun), http://nakedsecurity.sophos.com/2011/06/30/groupon-subsidiary-leaks-300k-logins-fixes- fail-fails-again/
42. The Wall Street Journal: Google Discloses Privacy Glitch (2009 Mar), http://blogs.wsj.com/digits/2009/03/08/1214/
43. Wang, W., Li, Z., Owens, R., Bhargava, B.: Secure and Efficient Access to Outsourced Data. In: Proc. of CCSW. pp. 55–66 (2009)
44. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In: Proc. of CCS. pp. 116–127 (2007)
45. Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F.: Improving Application Security with Data Flow Assertions. In: Proc. of SOSP. pp. 291–304 (2009)
46. Zavou, A., Pappas, V., Kemerlis, V.P., Polychronakis, M., Portokalidis, G., Keromytis, A.D.: Cloudopsy: an Autopsy of Data Flows in the Cloud. In: Proc. of HCII (2013)
47. Zavou, A., Portokalidis, G., Keromytis, A.D.: Taint-Exchange: a Generic System for Cross-process and Cross-host Taint Tracking. In: Proc. of IWSEC (2011)
48. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making Information Flow Explicit in HiStar. In: Proc. of OSDI (2006)
49. Zeldovich, N., Boyd-Wickizer, S., Mazières, D.: Securing Distributed Systems with Information Flow Control. In: Proc. of NSDI. pp. 293–308 (2008)
50. Zhang, Q., McCullough, J., Ma, J., Schear, N., Vrable, M., Vahdat, A., Snoeren, A.C., Voelker, G.M., Savage, S.: Neon: System Support for Derived Data Management. In: Proc. of VEE. pp. 63–74 (2010)
51. Zhu, D., Jung, J., Song, D., Kohno, T., Wetherall, D.: TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. ACM Operating Systems Review 45(1), 142–154 (2011)