SignedQuery: Protecting Users Data in Multi-tenant SaaS Environments

Eyad Saleh
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
eyad.saleh@hpi.uni-potsdam.de

Ibrahim Takouna
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
ibrahim.takouna@hpi.uni-potsdam.de

Christoph Meinel
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
christoph.meinel@hpi.uni-potsdam.de

***Abstract*—Software-as-a-Service (SaaS) is emerging as a new software delivery model, where the application and its associated data are hosted in the cloud. Due to the nature of SaaS and the cloud in general, where the data and the computation are beyond the control of the user, data privacy and security becomes a vital factor in this new paradigm. Several research studies reported that security and privacy are cited as the biggest concerns in adopting cloud computing. In multi-tenant SaaS applications, the tenants become concerned about the confidentiality of their data since several tenants are consolidated onto a shared infrastructure. Consequently, several questions raise, such as, how to ensure that tenant's data are only available to authenticated users? How to prohibit a tenant from accessing other's data? To address these concerns, we present *SignedQuery*, a mechanism designed to facilitate the process of securing data stored on the cloud. *SignedQuery* ensures data confidentiality by preventing any tenant from accidentally or maliciously accessing other tenants' data without breaking the functionality of the application. *SignedQuery* utilizes the usage of a signature to sign the tenant's request, so the server can recognize the requesting tenant and ensure that the data to be accessed is belonging to this tenant. *SignedQuery* intercepts the HTTP request objects at the tenant's internal network, create the signature and attach it to the request headers, then send the request to the SaaS provider where the signature is validated. We have successfully tested *SignedQuery* against OrangeHRM. The results showed that our approach is feasible, and incur a negligible overhead.**

I. Introduction

Software-as-a-Service (SaaS) has been growing rapidly over the last years and seems to be a promising software delivery model [1]. Gartner reported that SaaS is expected to have a healthy growth through 2015, where worldwide revenue is projected to reach $22 billion [2]. SaaS provides major advantages to both service providers as well as consumers. Service providers can provision a single set of hardware to host their applications and manage hundreds of clients (tenants). They can easily install and maintain their software. As for the consumers, they can use the application anywhere and anytime, they are relieved from maintaining and upgrading the software (on-premises scenario), and benefit from cost reduction by following the pay-as-you-go model [3].

Although SaaS offers several advantages to both service providers and users, such as the reduction of *Total Cost of Ownership* (TCO), better scalability, and better resource utilization, the users are still concerned about the security and privacy of their data. Privacy concerns exist wherever information about individual or organizations is processed and stored. Improper disclosure of information can be the cause of privacy issues. In 2009, Forrester Research reported that *Privacy and Security* has been selected by IT professionals as the primary barrier for not widely adopting the cloud computing.

Data encryption is a common approach to protect the confidentiality of user's data during transmission and storage [6], [7]. As for computation, a fully homomorphic encryption scheme has been introduced by [8] as a result of joint efforts between Stanford and IBM. This scheme supports computation over encrypted data. However, we believe that such a technique is still in the early stages of development, and would require huge extra cost. Another approach can be referred to as *information disassociation* [22], where the information is separated into parts, these parts are stored in geographically-distributed locations. Therefore, any party involved cannot benefit from the data it hosts. Recent interesting approaches such as *CloudProtect* [23] and *Silverline* [24] addressed the issue of data confidentiality from a different perspective. Both of them support keeping the data at the server-side confidential by encryption in a way that is transparent to the application. However, both of them require additional software to be installed or configuration to be made on the client machine, which prevents the user from using other computers to access his data, and hence, violates the basic concept behind the cloud.

In this paper, we introduce *SignedQuery*. A mechanism that provides an additional level of isolation among tenants' data by utilizing the concept of *Tenant Signature* without breaking the functionality of the SaaS application. *SignedQuery* enables the SaaS provider to limit the access to a tenant's data except for users who have access to the *Tenant-Key* and *Secret-Key*. Those keys should be available only to the legitimate tenant. *SignedQuery* is based on creating a signature for the tenant in his internal network using a secret key provided by the SaaS provider. This unique signature is used to sign all requests generated from a legitimate tenant by adding the signature to the request object. Thus, once the SaaS system receives a request, it validates the signature attached to ensure that the request is coming from a legitimate tenant that has privilege to access the data that belong to this particular tenant. Our approach is differs from existing security communication protocols, such as SSH or HTTPs. We do not intend to secure the communication. Rather, we want to ensure that the data

978-1-4673-6217-7/13/$31.00

of a certain tenant is accessible only to that tenant. However, there is no way to create a signature for the tenant at the database-level, since the query will be built and executed at the back-end, where the tenant does not have access. Thus, we decided to create the signature on the tenant's internal network, and add it transparently to the request object, and hence, pass it to the SaaS provider's side.

Our contributions are threefold: First, we introduce an approach to achieve data confidentiality without breaking the functionality of the application. Second, we present a technique that added another level of security by prohibiting a tenant from accessing other's data by using signatures. Finally, we implement our approach as a transparent HTTP proxy installed on the tenant's internal network, and evaluate it against OrangeHRM.

The rest of this paper is organized as the following: The introduction is followed by a discussion of the related work in Section II. We present an overview of *SignedQuery* and describe its architecture in Section III. In Section IV, we present the use case and our implementation. Section V discusses the experimental evaluation. Finally, we outline the limitations of our approach and conclude the paper in Section VI and VII respectively.

II. Related Work

Protecting users' data is an essential task in current systems. Researchers are proposing approaches and solutions to maximize the confidentiality of users' data. Social networks is considered an interesting area to study the impact of security and privacy issues on. FlyByNight[9] and Persona[14] are mainly designed to work with social networks, such as Facebook. They propose to store an encrypted version of the messages on Facebook's servers in away that is transparent to Facebook functionality. Thus, users will continue to use Facebook as usual while maximizing the level of their privacy. The main issue with such approaches is that they are designed specifically for social networks and cannot be applied to other domains. Several researchers [14], [15], [17], [18] propose solutions to perform some sort of processing on encrypted data, such as search and information sharing. However, all these approaches involves modifying the database layer as well as the application code, which is not the focus of our work, where we try to maximize the data confidentiality without changing/breaking the functionality of the application. Trusted cloud computing platform (TCCP) has been proposed by [16]. The idea is to provide a closed-box execution environment for the consumers, guaranteeing that the cloud provider cannot tamper with the users data. Moreover, it allows the consumers to remotely check whether the server is running a TCCP implementation or not. The main limitation of this approach is the infrastructure provider since they need to adopt TCCP first, then consumers are capable of using it. *CloudProtect* [23] and *Silverline* [24] are closely related to *HPISecure*. Both of them support keeping the data at the server-side confidential by encryption in away that is transparent to the application. However, *Silveline* proposed to dynamically analyze the application to determine which parts of the data can be functionally encryptable. It divides the users into groups, and assign a single encryption key to this group, facilitating encryption and information sharing at the same time. It assumes that any data is accessed by functions initiated by the user cannot be encrypted. In contrast to *Silverline*, *CloudProtect* encrypt all users' data and route users request to operate on it, if certain operations require data to be in plain text, it implements a protocol to expose this data for a short period of time. This would require an extra overhead, and so they introduced a relaxation policy to allow the user to trade-off security for performance. A tenant-oriented security-management architecture called TOSSMA has been proposed by Almorsy et al. [25]. TOSSMA enables SaaS providers to use it on the platform-level to serve several applications. As for the tenants, it allows them to define and customize the security requirement they need without (re)engineering the existing applications or write security integration code. Also, TOSSMA allows the tenants to define the security requirement on different levels, such as component, class, or method. The main difference to our approach is that TOSSMA is mainly targeted to securing the resources at the SaaS-side in general, and focus on the application code more than the database-level.

III. Overview of SignedQuery

The goal of *SignedQuery* is to improve the confidentiality of users' data stored on the cloud. We propose the usage of a signature to sign the tenant's request, so the server can recognize the requesting tenant and ensure that the data to be accessed is belonging to this tenant. *SignedQuery* uses a custom HTTP scheme based on a keyed *Hash Message Authentication Code* (keyed-HMAC) [12] for authentication. To create the signature, we concatenate selected elements of the HTTP request to form a string. Then, we use the tenant's secret key (provided by the SaaS provider) to create the HMAC of that string. When the system (server-side) receives a request; it validates the signature and either drop the request because of invalid signature or proceed further if the signature is valid.

We implemented *SignedQuery* on top of Fiddler [10] as an HTTP proxy installed on the tenant's internal network and SaaS provider's side. As for the use case, we decided to select Orange Human Resource Management (OrangeHRM) [11]. OrangeHRM is the world's most popular open source HRM application with more than one million users globally and more than 600,000 downloads.

A. Motivation Scenario

There are several database architectures that could be used for multi-tenancy, such as completely isolated database for each tenant, shared database with different schemas, or shared database and shared schemas. However, in this paper, we focus on the shared environments where several tenants are consolidated into a single database instance. The basic technique to accomplish data isolation is adding a tenant-id column to all tables, and then change the queries, functions, and triggers to add the tenant-id filter.

Consider "Al-Ghanem" and "ServTech", two industrial companies working in the construction field. Both companies use an on-demand ERP system (that utilizes that shared database approach) provided by a well-known SaaS provider. Both companies compete in the market, and therefore, the data of their personnel, projects, and financial records are of much importance. The data for both companies is at risk of being

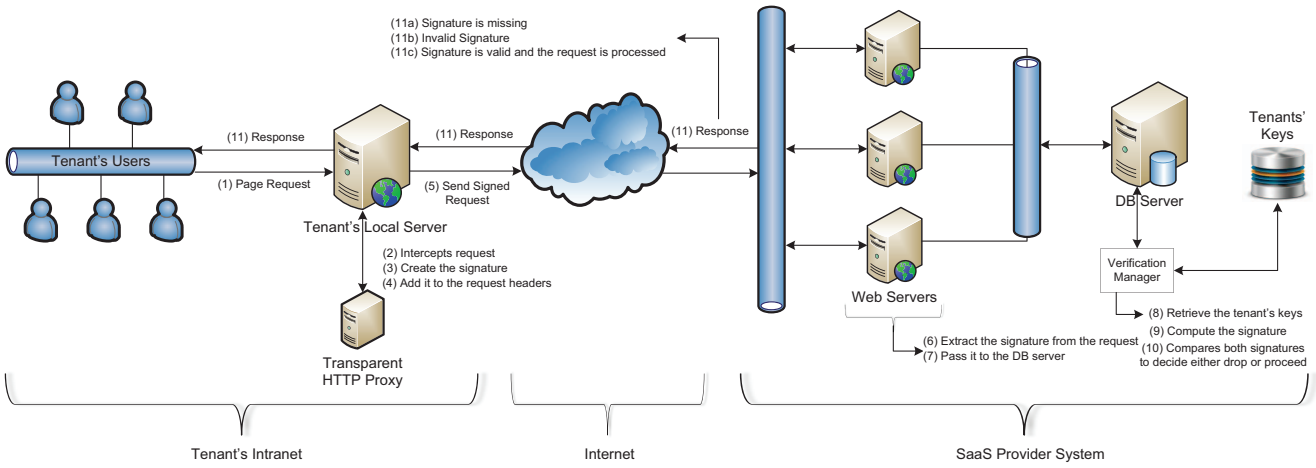

Fig. 1. SignedQuery Architecture

Algorithm 1 Signature Creation Algorithm

Input: $Tenant\text{-}Key$, $Content\text{-}MD5$, $Content\text{-}Length$, $Timestamp$, $HTTP\text{-}Method$, $HTTP\text{-}Request\text{-}URI$, and $Secret\text{-}Key$
Output: $Signature$ ($HMAC\text{-}SHA256$)

//Start forming the predefined-string ($PreStr$)
if $PreStr$!= $null$ **then**
  $PreStr = null$
end if
$PreStr$ = $Tenant\text{-}Key$
Concatenate ($PreStr$ and "$Newline$")
if $Content\text{-}MD5$!= $null$ **then**
  Concatenate ($PreStr$ and "$Content\text{-}MD5$")
end if
Concatenate ($PreStr$ and "$Newline$")
if $Content\text{-}Length$!= $null$ **then**
  Concatenate ($PreStr$ and "$Content\text{-}Length$")
end if
Concatenate ($PreStr$ and "$Newline$")
Concatenate ($PreStr$ and "$Timestamp$")

// Start forming the String-To-Sign (STS)
if STS != $null$ **then**
  $STS = null$
end if
STS = $HTTP\text{-}Method$
STS = concatenate (STS, "$Newline$")
STS = concatenate (STS, $HTTP\text{-}Request\text{-}URI$)
STS = concatenate (STS, "$Newline$")
STS = concatenate (STS, $PreStr$)

[$String\text{-}To\text{-}Sign$] ← UTF8-Encoding($STS$)
$Signature$ = $Base64$ ($HMAC\text{-}SHA256$ ($Secret\text{-}Key$, $String\text{-}To\text{-}Sign$))

exposed to third parties, either accidentally or maliciously. Accidentally, when for example a software bug or malfunction resulted in sharing one tenant's data with others, such as if the developer forget to filter a SELECT query by the tenant-id, this would result in returning the data of all tenants. Or maliciously, when an attacker exploits a weakness in the software stack and gain illegal access to the data, such as SQL-injection or Cross Site Request Forgery (CSRF) attacks. Therefore, our proposed approach is capable of solving such issues by preventing any tenant from accessing others' data through the utilization of tenant's signatures.

B. Architecture of SignedQuery

Typically, the tenant's users start using the SaaS application by accessing it through their web browsers; the requests are routed through the tenant's internal web server, where we implement our transparent proxy. The proxy intercepts the request, creates the signature using the tenant's *Secret-Key* provided by the SaaS provider, and adds it to the request. This process is transparent to the users, whereas only the internal web server is required to have the proxy and the *Secret-Key*, and hence, the users are allowed to use the SaaS application from any machine within the boundary of their internal network without any requirement of a special software to be installed. When the SaaS application receives the request, the signature is extracted and passed to the *Verification Manager*. We implement the *Verification Manager* as a transparent proxy between the application and the back-end database by utilizing the open-source software MySQL-Proxy [13]. MySQL-Proxy can monitor, analyze, and transform communication between the application and the back-end database. MySQL-Proxy allows intercepting, logging, filtering, and modifying the queries before execution. Following best practices, we propose to have two instances of the *Verification Manager* to maintain the goal of high availability and reliability, and to avoid the single point of failure, where the two instances utilize the fail-over technique. The architecture of *SignedQuery* is shown in Figure 1.

C. The Signing Process

Below we describe the steps we follow to sign the requests.

1) ***Request Signing****: SignedQuery* uses a custom HTTP scheme based on a keyed *Hash Message Authentication Code* (keyed-HMAC) for authentication. To create the signature, we concatenate selected elements of the HTTP request to form a string. Then, we use the tenant's *Secret-Key* (provided by the SaaS provider) to create the HMAC of that string. This HMAC is the *signature* used to authenticate the request. Finally, we add the signature to the request as a parameter of the headers section.

When the system (server-side) receives a request, it checks whether the request contains a signature or not. If the signature is not present in the request, then we assume that it comes from illegal source. Thus, we drop the request and send an error message to the requester explaining that the signature is not present in the request. On the other hand, if the signature is present, the process of validating the request starts by fetching the tenant's secret key. This secret key is used to compute the signature of the received request the same way the tenant did. Then, we compare the signature we just calculated with the one present in the request, if they match, then we conclude that the signature comes from the tenant who have access to the secret key, and therefore consider it as the authorized tenant to whom the key has been issued. If the two signatures do not match, the request is dropped and an invalid-signature error message is sent to the requester (tenant). Figure 2 shows the signature creation and validation process.

2) ***HTTP Authorization Header****: SignedQuery* utilize the standard HTTP *authorization header* to attach the signature of the tenant to the request. The authorization header has the following form:

Authorization: HPI TenantKey:Signature

Similar to the concept of asymmetric cryptography, every tenant is assigned a *Tenant-Key* and a *Secret-Key*. However, the *Tenant-Key* is not public, rather it is attached to the request in plain text. When the system receives a signed request, it uses the *Tenant-Key* to identifies the *Secret-Key* that was used to compute the signature at the tenant's side, which is indirectly identifies the tenant who issues the request.

We follow the RFC 2104 (HMAC-SHA256) [12] to create the signature. Thus, the signature will vary from request to request, even for the same tenant. If the signature calculated by the system matches the signature included in the request, then the requester (tenant) shows that he has access to the *Secret-Key* that has been issued to this tenant, and therefore can get access to the data belongs to that tenant. Algorithm 1 shows the steps we follow to construct the signature.

HMAC-SHA256 is an algorithm defined by RFC 2104. The algorithm takes as input two byte-strings, a key and a message. We use the Secret-Key that is provided by the SaaS provider as the key, and the UTF-8 encoding of the String-To-Sign as the message. The output is also a byte-string, called the digest. The *signature* part of the authorization header is constructed by Base64 encoding this digest. Figure 2 shows the workflow of creating the signature at the tenant's side and then validating it at the provider's side.

3) ***Construction of String-To-Sign****:* As the system compares the computed signature with the signature provided in the request, it is very important that computing the signature should follow the same method on both sides, the tenant side as well as the provider side

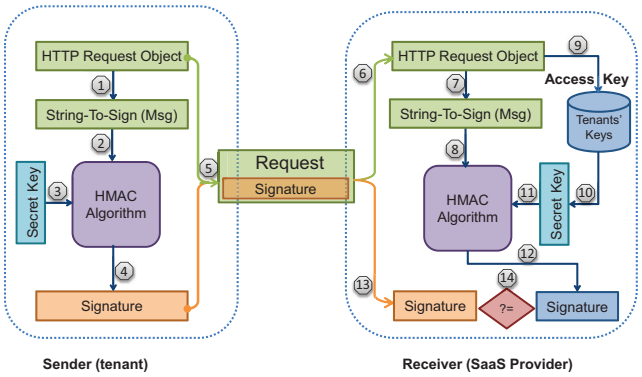

Fig. 2. Signature Creation and Validation Process

To construct the String-To-Sign, we first start by an empty string, then add the HTTP method, followed by the HTTP Request-URI. Finally, we add the predefined-string. The predefined-string consists of four parameters, the Tenant-Key, followed by the content-MD5 and content-length respectively. Finally, the timestamp is added at the end of the string.

The timestamp is a critical parameter in the String-To-Sign because it will help in creating a different String-To-Sign for every request, which will make generating such signatures by malicious users a very hard process. Moreover, the client-timestamp included with the request must be within 15 minutes of the SaaS provider's server time when the request is received. Otherwise, the request will fail and the system will drop it with an error message Request-Timeout. This would limit the possibility of intercepting and manipulating requests during transmission.

According to the Internet Engineering Task Force (IETF) [19], non-standard HTTP headers can be used following the format *x-name* = *value* (e.g., x-date=26/03/2013). Although it was deprecated as of June 2012 [20] for standardizing issues, we used it to add custom headers to the Request/Response HTTP objects. Therefore, we use the *x-hpi-date* non-standard header to include the timestamp of the request.

IV. Use Case and Implementation

Use Case: In Jan 2011, Forrester Research published a report with the title "Which Software Markets Will SaaS Disrupt?" [21], one of the conclusions was that Human Resource Management Software (HRMS) is a good candidate for SaaS. Thus, we decided to select an HRMS as our use case. Orange Human Resource Management (OrangeHRM) is the world's most popular open source HRM application with more than one million users globally and more than 600,000 downloads [11]. OrangeHRM released under the GNU General Public License and targets small and medium enterprises. The system functionality includes employee information management, absence and leave management, recruitment management, performance evaluation, etc. From technical point of view, OrangeHRM is implemented using PHP and MySQL, and its relational data model contains about 115 tables.

Implementation: We implemented *SignedQuery* on top of Fiddler [10] as an HTTP proxy installed on the tenant's internal network and SaaS provider's side, where the Request/Response objects of the HTTP protocol are intercepted, signature is authenticated, and proceed further according to the validity of the signature. To validate our approach, we extend Fiddler using .NET technologies. Fiddler [10] offers the possibility of intercepting requests before and after sending them to the server. It also offers the same for the response objects. Thus, by utilizing Fiddler, we are able to intercept and manipulate request/response objects when required. Furthermore, Fiddler offers the entire request/response objects and their data as a .NET objects that is accessible to any application that extends Fiddler, so extending Fiddler allows us to read all headers and body content of the request/response objects, add new content, change exiting content, and delete unwanted data.

V. Discussion

In this section we summarize our experiments to assess the applicability and feasibility of our approach. The discussion of the results is also presented.

Experiments: We develop *SignedQuery* as a proof-of-concept to validate our approach in enforcing security isolation between tenants. We tested our approach against OrangeHRM, the world's most popular open source HRM application. Figure 3 shows an example of an HTTP request with a signature added under the authorization header. We were able to sign the requests, validate it at the SaaS provider's side, and reject requests without signature or requests with invalid signatures.

We run our experiment on an Intel Core i5 2.66GHz with 4GB of Ram. We assume that the server utilization at the SaaS provider's side is below 70%, so we can ensure that response time is not affected by the server status. We make two identical sets of tenants; each set runs five simultaneous tenants and issue around 100 requests per set against OrangeHRM. The only difference between the two sets is that the first set issues requests with signature, while the second set issues requests without signature. We repeat the experiment (issue around 100 requests for every set containing five simultaneous tenants) three times, and take the average response time for every request.

Results: We reported the response time as depicted in Figure 4. It is clearly shown that the difference in response time for most of the requests is almost negligible. This indicates that the signing process does not incur an expensive overhead on the system. However, in some situations, particularly within the first 12 requests, the signing process causes extra overhead. We hypothesize this for the following reasons. First, requests from 1-6 are mainly inserting records in the database, while others are mainly fetching data. Second, requests 7 and 8 seem to be kind of inverted since the response time for the signed request is less than the unsigned one, the main reason is that the type of the query in those requests is viewing all employees for this tenant (i.e., *Select * from employee where tenant_id=x*), and since the unsigned request is executed before the signed one in our experiment, we believe that the database cached the result, so when the signed request execute the same query, the result retrieved from the cache, which is much faster than executing it against the database. To be more confident about

```
GET /orangehrm-2.7/symfony/web/index.php/pim/addEmployee HTTP/1.1
Host: 192.168.56.101
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:19.0) Gecko/20100101 Firefox/19.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.56.101/orangehrm-2.7/symfony/web/index.php/pim/viewEmployeeList/reset/1
Cookie: PHPSESSID=he2uc051elncqssshh3opvbeg7; Loggedin=True
Connection: keep-alive
x-hpi-date: 2013-03-18T18:18:34Z
Authorization: HPI
AKIAIM3ATN4SLQ2JLHEA:adg8XKmzvPk%2BZvD6IfoHzKlMyS0i%2BooyyNw%2FLJBl96A%3D
```

Fig. 3. Http Request With Signature

this hypothesis, we repeat the same experiment with the same setup, but we clear the database cache for every repetition. As expected, the signed requests require more time to respond for entire experiment. Since the Figure of the second experiment that shows the difference in response time is very close to Figure 4, except few requests, we decided to take it out.

Figure 5 shows the Inverse-Cumulative Distribution Function (ICDF) of our performance degradation. It clearly indicates that about 60% of the signed requests incur less than 5% of extra overhead in reference to the original response time without signature. It also indicates that more than 80% of the signed requests incur less than 20% of extra overhead in reference to the original response time without signature. On the other hand, less than 2.5% of the requests incur about 40% of extra overhead. As a result, we believe that our approach is capable of providing security isolation among tenants with an acceptable extra overhead for certain types of requests, such as inserting new records into the database.

VI. Limitations of SignedQuery

Anywhere, anytime. *SignedQuery* is currently implemented as a desktop application that needs to be installed on the web server of the tenant's internal network and on the SaaS provider's side. This deployment approach violates the concept of mobility available in the cloud. For example, if a user wants to access the SaaS application out of the tenant's internal network? Or if the user needs to use a tablet or a mobile phone? In such cases, the SaaS provider needs to ensure that the user would still be able to use the SaaS system. One approach would be that the users login into the tenant's network through VPN, and therefore be able to use the system. Another approach is installing a light-weight application on the tablet or mobile phone that is provided by the SaaS provider. This light-weight application is responsible for generating signatures and attaching them to the requests in a transparent manner. We will consider this also in our future work.

Security at the database-level. Securing data on the cloud involves several factors, such as cost, availability, and performance. Although we provide a security mechanism by using signatures. Some efforts still could be done on the database-level to enhance the level of security. This will be also part of our future work.

Document-Based Applications. The main target of *SignedQuery* is enterprise-applications, such as ERP, CRM, and HRM. Part of our future work is to extend *SignedQuery* to work on document-based applications, such as Google Docs.

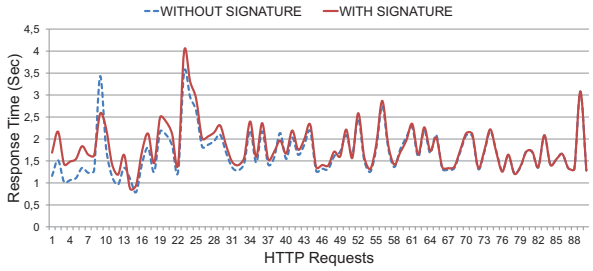

Fig. 4. Response time in seconds

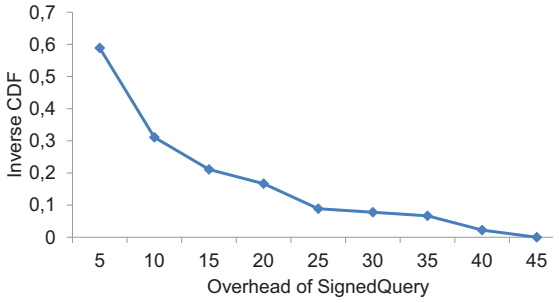

Fig. 5. Inverse CDF for the overhead of SignedQuery

VII. Conclusion

Data confidentiality is one of the key concerns in cloud computing. Organizations are not widely adopting the cloud because of issues related to security and privacy of their data. In this paper, we present *SignedQuery*, a mechanism designed to facilitate the process of securing data stored on the cloud. *SignedQuery* ensures data confidentiality by preventing any tenant from accidentally or maliciously accessing other tenants' data without breaking the functionality of the application. *SignedQuery* utilizes the usage of a signature to sign the tenant's request, so the server can recognize the requesting tenant and ensure that the data to be accessed is belonging to this tenant. *SignedQuery* intercepts the HTTP request objects at the tenant's internal network, create the signature and attach it to the request headers, then send the request to the SaaS provider where the signature is validated. We have successfully tested *SignedQuery* against OrangeHRM and proved that our approach is feasible. There are still open challenges that we plan to cover in the future work, such as extending *SignedQuery* to work with document-based applications and also extending *SignedQuery* to provide a technique to protect the data at the database-level.

Acknowledgement

The authors would like to thank Mohammad AbuJarour from SAP for his valuable insights and feedback.

References

[1] A. Konary, S. Graham, and L. Seymour: *The future of software licensing: Software licensing under siege*. International Data Corporation, White Paper, 2004.

[2] Gartner website: *Software-as-a-Service Revenue*. [Online]. Available: http://www.gartner.com/newsroom/id/1963815 [retrieved: Mar, 2013]

[3] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and M. Zaharia: *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report, University of California, Berkeley, USA, 2009.

[4] B. R. Kandukuri, R. P. V., and A. Rakshit,*Cloud security issues*. In Proc. of 2009 IEEE International Conference on Services Computing, Bangalore, India, 2009.

[5] T. Espiner: *Data is more secure in the Cloud*. [Online]. Available: http://www.zdnet.com/google-data-is-more-secure-in-the-cloud-3039854667/ [retrieved: Mar, 2013]

[6] C. Wang, Q. Wang, K. Ren, and W. Lou: *Ensuring data storage security in cloud computing*. In Proc. IWQoS 09, Charleston, South Carolina, USA, 2009.

[7] A. Saha: *Computing on encrypted data*. In Proc. ICISS 2008, Hyderabad, India, 2008.

[8] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford, 2009.

[9] M. M. Lucas and N. Borisov. *Flybynight: mitigating the privacy risks of social networking*. In Proc. of ACM WPES, Virginia, USA, 2008.

[10] Fiddler website. [Online]. Available: http://www.fiddler2.com [retrieved: April, 2013]

[11] OrangeHRM Website. [Online]. Available: http://www.orangehrm.com [retrieved: April, 2013]

[12] HMAC RFC 2104 Website. [Online]. Available: http://tools.ietf.org/html/rfc2104 [retrieved: April, 2013]

[13] MySQL Proxy Website. [Online]. Available: http://dev.mysql.com/downloads/mysql-proxy [retrieved: April, 2013]

[14] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Strain. *Persona: An online social network with user-defined privacy*. In Proc. of ACM SIGCOMM, Barcelona, Spain, 2009.

[15] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. *Public key encryption with keyword search*. In proc. of Eurocrypt, Interlaken, Switzerland, 2004.

[16] N. Santos, K. P. Gummadi, and R. Rodrigues. *Towards trusted cloud computing*. In Proc. of HotCloud, San Diego, USA, 2009.

[17] H. Haclgümüs, B. Lyer, C. Li, and S. Mehrotra. *Executing SQL over encrypted data in the database-service-provider model*. In Proc. of ACM SIGMOD, Wisconsin, USA, 2002.

[18] H. Wang and L. V.S. Lakshmanan. *Efcient secure query evaluation over encrypted XML databases*. In Proc. of VLDB, Seoul, Korea, 2006.

[19] IETF Website. [Online]. Available: http://www.ietf.org/ [retrieved: April, 2013]

[20] RFC 6648 Website. [Online]. Available: http://tools.ietf.org/html/rfc6648 [retrieved: April, 2013]

[21] A. Bartels, L. Herbert, C. Mines, and Sarah Musto: Forrester Research. [Online]. Available: http://www.forrester.com/Which+Software+Markets+Will+SaaS+Disrupt/fulltext/-/E-RES57405 [retrieved: April, 2013]

[22] K. Zhang, Y. Shi, Q. Li, and J. Bian. *Data Privacy Preserving Mechanism based on Tenant Customization for SaaS*. In Proc. IEEE MINES, Wuhan, China, 2009.

[23] M. H. Diallo, B. Hore, E. C. Chang, S. Mehrotra, and N. Venkatasubramanian. *CloudProtect: Managing Data Privacy in Cloud Applications*. In Proc. IEEE Cloud, Hawaii, USA, 2012.

[24] K. P. N. Puttaswamy, C. Kruegel, and B. Y. Zhao. *Silverline: toward data confidentiality in storage-intensive cloud applications*. In Proc. of ACM SOCC, Cascais, Portugal, 2011.

[25] M. Almorsy, J. Grundy, and A. S. Ibrahim. *TOSSMA: A Tenant-Oriented SaaS Security Management Architecture*. In Proc. of IEEE Cloud, Hawaii, USA, 2012.