THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,
Volume 16, Special Issue 2015, pp. 345-356

A SECURITY PARADIGM FOR PAAS CLOUDS

Mehmet Tahir SANDIKKAYA, Ali Emre HARMANCI

Faculty of Computer and Informatics, Istanbul Technical University, Turkey
E-mail: sandikkaya@itu.edu.tr

The paper proposes a novel comprehensive security paradigm for PaaS clouds where the customer programs are treated as separate processes instead of web service threads, which is the current method. The cloud customer has more flexibility on her programs and data on this new approach. The programs, together with their respective data, are isolated within process containers, which are encrypted storage entities carefully designed to enforce access control, secrecy, integrity and service level agreements.

Key words: PaaS, security, cloud, process container, isolation, process-based.

1. INTRODUCTION

Since the emergence of the cloud concept, Platform-as-a-Service (PaaS) stands out with unprecedented service model; therefore, specific security requirements. In the PaaS service model, cloud customers prepare or obtain programs and offer them to their users through a cloud provider's infrastructure. The main benefit of this model for the customer are to decrease the cost and to avoid continuous maintenance of the infrastructure by outsourcing. Besides, the cloud provider achieves efficiency by pooling the resources. Numerous customer programs reside in a few number of provider hosts simultaneously to reduce the overall costs.

The novel concerns about PaaS security arise from two issues: outsourcing and multi-tenancy. In an outsourced service model where unknown programs run on unknown infrastructure, both the programs and the underlying infrastructure requires mutual guarantees against misbehavior. Together with that, multi-tenancy necessitates appropriate isolation in between programs that run on the same hardware. Additional to PaaS specific requirements, the provider must help to fairly manage access control on behalf of the customers and to build a standard computing interface to avoid provider lock-in.

There are many commercial PaaS instances in common use today [1–4]. Unfortunately, with respect to the mentioned security concerns, these instances do not satisfy expectations. According to the service level agreements (SLAs), there are no mutual guarantees for misbehavior. Especially, the customer does not have any applicable rights to protect her programs or data.

Program isolation is also problematic. In these instances, the computational power is provided through threads to the customers' programs. Therefore, the isolation has to be fulfilled in between threads. However, thread isolation has several known security drawbacks [5–7].

Access control is generally not built-in and totally left to the customer programs' logical checks. Much-praised role-based access control (RBAC) is not suitable for cloud as data exposure cannot be controlled when it is once disclosed to the access control authority [8]. Moreover, trust relations are blurry in the cloud and it is not clear if the provider is trusted as an access control authority.

Finally, it is known that, most of the time, providers build systems to lock users in their infrastructure with specific interfaces or mechanisms, not the other way around.

The aim of the paper is to mitigate the aforementioned security problems by proposing a common container design specific to process-based PaaS. It is believed that process-based design better fits PaaS and has better security qualifications. A common container design may ease mutual guarantees, fine-grained

customer/program/data-specific settings, as well as enforce user authentication and authorization with convenient encryption techniques.

The rest of the paper is organized as follows: In the following section the current state of the PaaS clouds in use is discussed together with the related work. The contribution of the paper is also introduced in this section as a comprehensive integrated security solution. The third section is spared for the details of the proposed security paradigm. It starts with the design of the complementary components at the provider's hosts, continues with the design of the process container and the details of the cryptologic mechanisms build on the process container. Fourth section mentions optional or future protection mechanisms that can be build on top of the proposed design as it states possible directions to hide data or program functionality from the untrustworthy provider. The fifth section concludes the paper.

2. RELATED WORK AND CONTRIBUTION

In the following sections the related work is given in the sense of methods of providing computation. Then, the paper's contribution, which is conceptually mentioned in a previous work [9], is refined and presented as an integrated security solution.

2.1. Threads vs. processes as units of providing computation

The computation may be provided by two means in practice. The customer may use the provider's computational power either as a separate process in the operating system or as a thread within a specific process of the operating system. Threads co-exist together with other threads within the process that contains them. On the other hand, a process is a collection of executable code and related data to perform a given task in its own dedicated context. Historically, threads co-exist with processes to overcome the computational burden of inter-process communication, which is useful for parallel processing and multitasking. In an operating system where threads do not exist, processes must communicate through shared objects such as files, sockets or shared memory areas which are controlled by the operating system. On the other hand, threads reside in the context of the same process; therefore they can signal each other independently from the operating system. Actually, as all of the threads of a process maintain the same memory area as a whole, each thread can access each other's variables, or even stacks [5].

Utilizing threads as many of the commercial PaaS deployments [1 – 4] may have serious security drawbacks. Even though several studies have been made to isolate threads [6, 7, 10–14], none of the referred studies properly isolate all different kinds of resources (file, network, processor and memory) that an executable code can access. File and network access isolation is quite straightforward with the help of software abstractions and managed by every approach. Processor usage is controlled and managed statistically in Sandikkaya et al.'s study [7]. Memory isolation is achieved up to some extent by mimicking a virtual operating system on top of the Java Virtual Machine (JVM) [15] in KaffeOS [11]. Other approaches that help to isolate threads generally limit the communication capabilities of threads to prevent information leakage. As a result, each step to further isolate a thread makes it more similar to a process and less performing as a thread. According to Tanenbaum [5], it is neither possible nor necessary to isolate threads.

On the other hand, utilizing processes in PaaS clouds is beneficial according to the given definition by two means. The functional benefit is the fact that processes are more suitable to encapsulate parallel logic than threads. The security benefit is the ease of isolation. By definition, the operating system executes each process in its own context isolated from others. As a matter of fact, a process better suits a PaaS cloud than a thread for providing the computation.

2.2. An integrated security solution

When a program is run locally, the operating system uses a process to set its memory space, data, configuration and user. These four are clear responsibilities to the operating system when programs run locally, yet undefined when programs run in the cloud. However, they must be set prior to switch any executable binary to running state. Handling of these responsibilities in the cloud are discussed below.

The memory is always under control of the operating system as a physical resource. Therefore, the responsibility of defining the memory space for a program always belongs to the operating system where the

program resides. While running programs in the cloud, the operating system must isolate the memory space according to the requirements of the customers. A process virtual machine, together with complementary software components, are set up as software trusted computing base (TCB) [16] for each process. The TCB is capable of setting up proper memory space before execution with respect to the users.

The program could be run with some data to process. This data is desired to be in close vicinity with the program for ease of access. In that case, packaging the program and the related data together in a container and treating them as a single conceptual object in the cloud is beneficial. Such a structure is introduced and named as the process container.

User management is a difficult area of cloud. Users are limited in local operating systems and each user runs her programs in her context. On the other hand, the number of customers or users is unlimited in the cloud and no dedicated context exists. Moreover, the program to be run may be initiated by a first time user who just migrated. As a result, the cloud is a system where every customer or user can receive an execution service from every registered host according to their SLAs. This functionality must be built-in to the solution. A convenient approach is to carry related userspecific settings of a program together with it.

A program running in the cloud may receive configuration settings from three sources. The first one is the SLA. The customer may propound her needs and the provider may limit them if she thinks that they may harm the cloud infrastructure. Probably, at the end the cost is negotiated or a standard agreement is signed. The second one is the configuration that is required to run the program appropriately. The network access to another service or a file access might be necessary. This kind of configuration is defined by the programmer. The third set of configuration is defined by the user while initiating the program.

Either the cloud customer as she is the owner of the program or one of the users authorized by the customer can run programs in the cloud. A remote connection is made to the host and a command is sent on how to initiate the program. There is a software component that handles remote connections, enforces policies and permissions and initiates program execution on each host, which is called policy enforcement point (PEP). After receiving the command, the PEP ensures the access rights and logs the access with an undeniable secure logging mechanism [9]. The access rights are read through stuck policies [17] on the process containers. Then, the PEP configures a TCB with respect to the effective permissions and executes the program on top of the TCB; therefore leaving the program logic to the customer program but controlling its resource access via TCB. Each process' input, output and error streams are attached to the remote user, so the process is controlled by the user during its lifetime. A conceptual drawing of the outlined scheme is given in Figure 1.

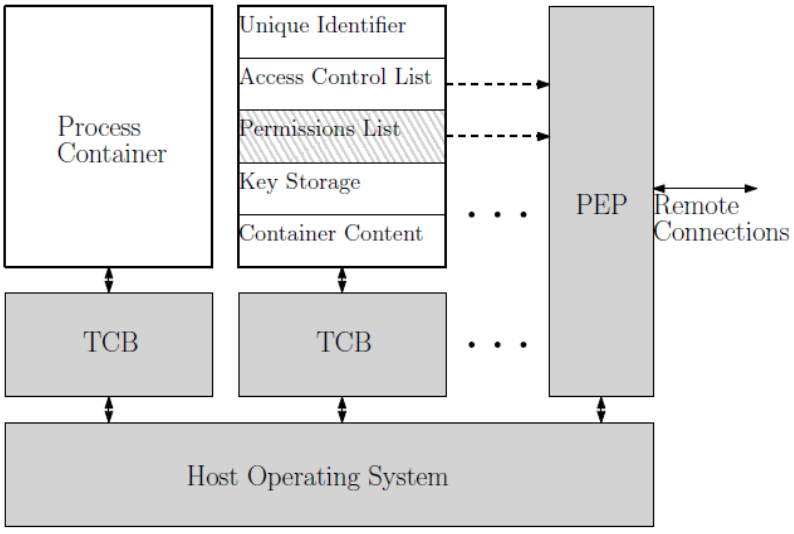

Fig. 1 – Two process containers in a host of a cloud provider are shown. The five areas of rightmost one are sketched in detail. Cloud provider's components are colored in gray. The containers are prepared by the cloud customers, however the shaded field is signed by the provider to ensure the compliance with the SLAs.

3. THE DESIGN OF THE PROPOSED SECURITY PARADIGM

The proposed security paradigm's design is explained briefly in four parts. In the first part, the complementary components at the provider side is described. After briefly mentioning the scenario that will be used in the paper, the third part describes the process container. In the fourth part, the cryptologic structures used in the process container is explained.

3.1. The design of the host components

Two components, the TCB and the PEP, reside on top of the operating system to help a process container to function in each host of a cloud provider. The design of these components are explained in this section before going further into the details of the process container.

3.1.1. The trusted computing base

The trusted computing base is a layer of software that lays on top of the host operating system to run customer programs. It acts also as a mediator in between the customer programs and the operating system to manage the resource access. Each operating system resource is abstracted as a software class in the trusted computing base. The customer programs' sole access to the resources is through these classes.

The aim of introducing a trusted computing base is twofold. The first benefit is providing a standard programming environment which helps to prevent provider lock-in. The second and the most important benefit is regulating program's access to the operating system resources by properly inspecting each access even before assigning the resource to the program. The permissions a program held are kept stuck on the process container. The sticky policies are read and the TCB are configured accordingly by the PEP before programs' execution.

The TCB concept is materialized with Java Virtual Machine. This decision is made as Java [18] is a common standard programming language. Note that, the presented ideas are independent from the programming language and adaptable to other languages as well. It can be claimed that offering a standard Java Virtual Machine where standard Java programs can run on as processes enables inter-cloud migration, thus against provider lock-in.

TCB is presented to be configured by the PEP. The configuration is done per-user before each execution. Hence, the effective permissions can be set accordingly at each access for each separate user by the TCB.

3.1.2. The policy enforcement point

The policy enforcement point is a communication access point for process containers that enforces access rights and permissions. The PEP has two responsibilities. The first one is to determine the permissions of the customer programs currently running on the host and configure the respective TCB instances according to the permissions stated by the stuck policies of the process containers. The correctness and fairness of the permissions are ensured by cryptologic means and explained later in Section 3.3.

The second responsibility of the PEP is to manage the remote connections based on the access rights also stated in the stuck policies on the process containers. Similarly, the correctness of the access rights is ensured by cryptologic mechanisms. Upon receiving an access request from a user, the PEP polls the host to check whether the container exists on the host, then if it exists, the PEP checks the access rights to verify if the requesting party has required access rights. If these checks are passed, the PEP logs this access and allows the requesting user to access the process container. Note that, even after PEP grants access to a user, she may require additional cryptologic keys to acquire meaningful information. Such details are covered in the sections 3.3 and 3.4 where the design of the process container and the cryptologic enforcement mechanisms are discussed.

The PEP observes the second and third areas of the process container as discussed in section 3.3. The second area includes information on who can access the process container. According to this information, PEP regulates access from outside the cloud. The third area includes information on how the process

container interacts with the host operating system. The PEP configures the TCB based on that information before each user access, so that the permissions are rendered effective before program execution.

3.2. A practical example

It is thought that exemplifying a practical instance of a process container may help to further explain the concepts within the design. Imagine a university intends to standardize the grading of the students. To enforce it, the university implements a generic course grade calculation program that accepts homework and examination grades as inputs and produces final grades as outputs. An instructor is responsible to feed the examination grades as a file and a teaching assistant is responsible for the homework grades. The students may read their grades any time. Obviously, the calculation takes place in the cloud.

In such a setting, there are one program field and three data fields in the content area of the process container. The instructor must have *read/write* access to the examination grades. The teaching assistant must have *read/write* access to the homework grades. The students require just *read* access to their grades. After the inputs are supplied, the instructor must have rights to execute the grade calculator program to produce the final course grades.

3.2.1. Having multiple process containers

The organization of the data and the programs in the process container and the number of process containers a customer may have are left to the customer. The customer may choose to own just one process container and place all of her programs in it with appropriate access rights and permissions. Alternatively, each container may include just one piece of data or program and permissions are set to build communications in between these containers. The presented design supports both extremes; however, it is believed the actual process containers include a few related programs and their respective data together to achieve a purposeful task. No contraints exist to have two containers with the exact contents or run them simultaneously if they are named differently.

In the above example, for the sake of simplicity, it is assumed that the university provides a separate process container for each separate course. Data access is also handled by a generic built-in program that is accessible by every user.

3.3. The design of the process container

The process container is prepared by the customer and it runs in one of the provider's hosts after its resource utilization policy is confirmed by the provider. It does not include any active decision making mechanism to prevent external intervention. Therefore, the process container structure is thought where only passive cryptologic protection could be effective.

The process container consists of five areas. The first two of them are kept in plain text, but signed to be protected against modifications. The third area is just for the use of the provider and it may contain partial information from the SLA. Therefore, it is encrypted with the public key of the provider. Moreover, it is signed both by the customer and the provider to indicate the agreement on the permissions. The fourth area contains encryption keys prepared according to the configuration set in the second area. In the fifth area the actual content is stored. It is considered as it may contain several fields where different access settings may applied.

The first area of the container contains a unique string signed by the customer to uniquely identify it in the cloud. An example of the first area is shown below. The standard Java package naming convention is followed in the example. σ indicates the signature operation and "ITU" stands as the customer.

$$\{ ltr.edu.itu.cloudapps.studentgrading.v1\_7\_73 \}\sigma_{ITU}$$

The second area includes the access control list (ACL) of the users to the contents of the container. This ACL is determined and signed by the customer. An example of this area is shown in Table 1.

In Table 1 the second area of the process container includes the ACL. This area is signed by the customer. The contents are stored in four fields, where field0 contains the program "grade.jar". Mr. Inst, is the only one that has rights to execute the program.

Table 1

	$field_0$ grade.jar	$field_1$ exam.txt	$field_2$ hw.txt	$field_3$ final.txt
Mr. Inst	X	R/W	R	R
Ms. Asst	-	R	R/W	R
[Students]	-	R	R	R

σ_{ITU}

The third area includes the permissions list. These permissions are effective in the host operating system when the programs are executed. Every user that is mentioned in the ACL must be stated in the permissions list as the related program may be initiated for that user at some point during the life cycle of the container. There may be generic permissions which are valid for every user such as reading data input or library files. Moreover, some other permissions may be added to the permissions list that is not visible in the ACL but required to execute the program correctly. A sound example is a network connection permission to a database which is necessary to collect some information during the computation. Another use of this area may be limiting the use of the programs according to the SLAs. This area is encrypted with the public key of the provider. After the encryption, the area is signed both by the customer and the provider.

Table 2

Mr. Inst :	~/data/exam.txt	read, write
	~/data/hw.txt	read
	~/data/final.txt	read, write
	~/programs/grade.jar	execute
	~/programs/grade.jar	bind, 80
	~/programs/grade.jar	listen, 160.75.*
	~/programs/grade.jar	maxMemory, 512M
	~/programs/grade.jar	maxDuration, 3'
Ms. Asst :	~/data/exam.txt	read
	~/data/hw.txt	read, write
	~/data/final.txt	read
[Students] :	~/data/exam.txt	read
	~/data/hw.txt	read
	~/data/final.txt	read

$\epsilon_{Provider}\sigma_{ITU}\sigma_{Provider}$

In Table 2 the third area of the process container includes the permissions. This area maps the ACL to the operating system permissions. Moreover, it may include additional necessary permissions or limitations dictated by the SLA. $\mathcal{E}$ stands for encryption.

Table 2 shows an example of the third area of the process container. In the example Ms. Asst's and Students' access rights are directly mapped to the operating system permissions as they do not execute any programs. However, permissions are not directly mapped for Mr. Inst. For example, it is supposed that additional permissions are required to communicate during execution, which is not unfamiliar for a program running in the cloud. The communication is limited to port 80 and the IP range starting with "160.75". Meanwhile, the program's memory and lifetime is limited according to the predetermined SLA in between the customer and the provider. An interesting extension is the write permission for final grades. This permission is necessary during program's execution while generating the output file.

The fourth area is designed as a key storage (see Table 3 below). It contains intermediate keys that are used to encrypt or sign the actual content stored in the fifth area. The intermediate keys are encrypted with the actual keys of the users or other related parties who have rights to access the programs or data within the container. Details of this area are covered in section 3.4.

The fifth area is the actual storage area of the container (See Table 4 below). It may contain sub areas to store several data or program *fields*. It is expected that the activities executed on this area stay in accordance with the access rights stated in the ACL. Such a control could be conducted by the operating system which is under supervision of the cloud provider. However, even the cloud provider does not conduct this control or even an adversary by-passes this control by some means, existence of cryptologic mechanisms to protect the secrecy and integrity of these fields increases trust to the cloud. It is possible to keep the data or programs secret, or at least it is possible to detect undesired modifications by cryptologic means. Details of this area are covered in section 3.4.

3.4. Design of the cryptologic mechanisms

Whenever the customer of the process container decides who and how can access the contents, the supporting cryptologic mechanisms help to enforce the customer's decisions. The design of these mechanisms is shown in the following sections.

3.4.1. Types of access

Process container content may be of two different types: program or data. According to this, there may be at most three different kinds of access types. If the accessed field is of type program, a user may either have execution rights or not. If the accessed field is of type data, a user may either have *read* access or *read/write* access or none.

Apart from those, a *write* access for the program field may be discussed. However, such an access implies a modification in the program itself, therefore a different program. This kind of access may be convenient for versioning. Still, it is intentionally left out of the scope of the paper. The customer may prepare a new process container with the new version of the program and deploy it on the cloud.

3.4.2. Mapping the access rights to encryption and signatures

Cryptologically preventing someone from reading some bit string (either program or data) is straightforward and known for decades; encryption. However, there is not any cryptologic mechanism to prevent someone from modifying a bit string. The only known mechanism is to discourage adversaries to not to modify the bit strings by utilizing signatures.

In the proposed process container, the mentioned access rights are mapped to encryption and signature schemes. The *read* and *execute* rights are mapped to encryption to enforce the settings crytologically. Similarly, *read/write* rights are mapped to signatures. Note that, it can be claimed that the *execute* right may be merged with the *read* right as both of the access types requires nothing more than reading the bit strings from a cryptologic perspective.

3.4.3. Mapping the fields to the users and roles

After mapping the access rights to the cryptologic primitives for each field, intermediate keys are formed. These keys cannot be sent immediately to the related parties for a few reasons. The management could be cumbersome, the related parties may be unknown at the time, related parties would not want to keep track of the keys or secure channels may not exist. As a result, it is reasonable to keep the intermediate keys on the process container. The intermediate keys are stored in the fourth area on the container and they are encrypted with the keys of their respective users.

Determining the users of the intermediate keys is the responsibility of the customer. Fortunately, this is a straightforward process as the subjects of the access rights are already defined. The generated intermediate keys for encryption or for signature must be encrypted with the public keys of the related subject; therefore, only the related subject can decrypt and access the related intermediate keys. When the subject is a principal user, this kind of key storage does not cause any complexity. However, when the

subject is a group of people defined by their roles of using the program and data in the process container, storing encryption keys may require additional cryptologic structures.

Attribute based encryption (ABE) is a recent cryptologic structure. Its first practical implementation is shown by Bethencourt et al. in 2007 [19]. ABE enables the encrypting party to encrypt the given plain text based on the previously determined attributes. For instance, it is possible to encrypt a text that can be decrypted just by people whose name is "Charlie Brown" or their management level is "chief". In the given example, the university may define a university member type which can be one of the "instructor", "teaching assistant", or "student". Similarly, it is possible to define many other attributes such as, "class of 2015", "computer science students", "women in engineering", and so on.

Three intermediate keys are formed for each field. One of them is a symmetric key used to encrypt and decrypt the field. The other two is a key pair used for signatures. The field must be signed after encryption. If a user has *read* or *execute* access, she needs the public key for the signature scheme to verify the integrity and the symmetric key to decrypt the field. On the other hand, when a user has *read/write* access, she needs all three of the keys. The intermediate keys with respect to the access types are encrypted either with the public key of a user or a group of users according to their attributes.

If a malicious user manages to by-pass the provider's *read/write* access control check, she may modify the contents of a field in the process container. However, even if it is possible to modify the content, the modified content is detectable by signatures. In that case, a secure logging mechanism is convenient to discover the last accessed user before modification.

Table 3

$$\text{ITU}: \left\{ \begin{array}{l} EncKey_0 \\ SigPubKey_0 \\ SigPrivKey_0 \\ EncKey_1 \\ SigPubKey_1 \\ SigPrivKey_1 \\ EncKey_2 \\ SigPubKey_2 \\ SigPrivKey_2 \\ EncKey_3 \\ SigPubKey_3 \\ SigPrivKey_3 \end{array} \right\} \epsilon_{ITU} \qquad \text{Provider}: \left\{ \begin{array}{l} SigPubKey_0 \\ SigPubKey_1 \\ SigPubKey_2 \\ SigPubKey_3 \end{array} \right\} \epsilon_{Provider}$$

$$\text{Mr. Inst}: \left\{ \begin{array}{l} EncKey_0 \\ SigPubKey_0 \\ EncKey_1 \\ SigPubKey_1 \\ SigPrivKey_1 \\ EncKey_2 \\ SigPubKey_2 \\ EncKey_3 \\ SigPubKey_3 \\ SigPrivKey_3 \end{array} \right\} \epsilon_{Mr.Inst}$$

$$\text{Ms. Asst}: \left\{ \begin{array}{l} EncKey_1 \\ SigPubKey_1 \\ EncKey_2 \\ SigPubKey_2 \\ SigPrivKey_2 \\ EncKey_3 \\ SigPubKey_3 \end{array} \right\} \epsilon_{Ms.Asst} \qquad \text{[Students]}: \left\{ \begin{array}{l} EncKey_1 \\ SigPubKey_1 \\ EncKey_2 \\ SigPubKey_2 \\ EncKey_3 \\ SigPubKey_3 \end{array} \right\} \epsilon_{[Students]}$$

In Table 3 the fourth area of the process container is the key storage. It includes the encrypted immediate keys. The indices of the intermediate keys correspond with the indices of the fields.

According to the given explanation in the above sections, the key storage can be formed as in Table 3. The customer has a copy of all keys; in this manner, access rights can be reorganized whenever needed. Mr. Inst has all the signature public keys and encryption keys as he has read access to each field, therefore he can both check the integrity of and decrypt these fields. He also has the signature private key of $field_1$, therefore he can change the contents and also sign the field after modification. Signature private key of $field_3$ is not required for Mr. Inst's access, but required while the program is executed to write the calculated grades to the output file. Ms. Asst has all the keys for $field_2$ as well as public signature keys and encryption keys for data fields to be able to view content. Students, as a group who has the attribute of being a student member of the university, can check the integrity of the data fields and decrypt these fields. The provider cannot

access any field, but check the consistency of the fields at any time for logging purposes. In this way, the provider can follow the illicit modifications.

The content area of the process container appears as in Table 4. This structure is quite straightforward. Each field is encrypted by the respective intermediate encryption key and then signed by the respective intermediate signature key.

Table 4

$$\begin{cases} \tilde{}/\text{programs/grade.jar} \end{cases} \epsilon_{EncKey_0}\sigma_{SigPrivKey_0}$$
$$\begin{cases} \tilde{}/\text{data/exam.txt} \end{cases} \epsilon_{EncKey_1}\sigma_{SigPrivKey_1}$$
$$\begin{cases} \tilde{}/\text{data/hw.txt} \end{cases} \epsilon_{EncKey_2}\sigma_{SigPrivKey_2}$$
$$\begin{cases} \tilde{}/\text{data/final.txt} \end{cases} \epsilon_{EncKey_3}\sigma_{SigPrivKey_3}$$

In Table 4 the fifth area of the process container includes the encrypted and signed content.

4. OPTIONAL OR FUTURE PROTECTION MECHANISMS

The proposed paradigm hides the data or programs by encrypting the fields. However, in the proposed scheme, when a user wants to decrypt one of the fields, the decryption should take place in the cloud. Even the data or programs stay decrypted for a temporary period of time, this may seem problematic when the provider is untrustworthy. It should be noted that, the customer trusts the provider to some extent and deploys her assets on the provider's infrastructure under these conditions. The provider serves the customers, but at the same time she is curious and tries to learn as much information as possible in case she is given opportunity. This adversary model is known as honest-but-curious adversary model.

The customer modestly wishes to hide her data from curious eyes, including the provider. Her data may be strictly personal or may be a valuable trade secret. Furthermore, the customer also wishes to keep her program's execution details obscure. Her program's algorithm may be a patented method that must be protected legally.

There are cryptologic structures to protect the customer from a honest-but-curious provider. Still, they are not mentioned directly in the process container structure intentionally as they are not currently matured enough to perform in a realistic setting. Some homomorphic arithmetic operations are known to be applicable on some public key cryptographic systems. However, each cryptographic system allows only one homomorphic arithmetic operation on encrypted data; either addition (e.g. Paillier [20]) or multiplication (e.g. ElGamal [21]). This fact limits the capability of generating useful programs even if this approach is relatively performing.

Fully homomorphic encryption (FHE) is a recent improvement where both addition and multiplication operations can be conducted on encrypted data [22]. Theoretically, it is possible to define Turing-complete programs with FHE even without decrypting the encrypted data. However, many problems still exist. First, both the computational and storage cost of FHE is currently colossal. Even optimized implementations are slower than regular computations up to eight to nine orders [23]. Second, programs written in any high-level programming language cannot be compiled or interpreted to programs that can use fully homomorphic encrypted data automatically. In a study [24], two widely-known algorithms are converted as a proof of concept. In a separate research trail, two studies [25, 26] solve the issue of determining the end of converted programs and offer a command set of assignment, addition, less-than comparison, goto, bitwise shift, bitwise AND, and bitwise XOR in C language. Yet, it is not expected that a software engineer to write programs for the cloud with such limited vocabulary or with a few converted algorithms. Third, even the performance and conversion issues are assumed to be solved, the data can be hidden by FHE, but the algorithm cannot. For example, if the customer wishes to protect the algorithm of her program, FHE is still useless if the provider carefully observes the CPU instructions and memory access patterns.

Indistinguishability obfuscation [27] is a very recent idea that even its implementation does not currently exist. In this approach, some core part of the program can be "punctured out" to be hidden from the adversary. As a result, theoretically, if the algorithm of the program is located in that specific puncture area, it could be possible to hide the algorithm. This idea can be used to hide the details of the functionality of the customer programs from the provider in the future cloud deployments if there will be implementations with reasonable time and storage costs.

5. CONCLUSION AND FUTURE WORK

The paper outlines the current PaaS environment and addresses its common functional and security problems. The roots of some of the security problems lie in the current approach to the PaaS concept. It is discussed that building more secure PaaS infrastructures is possible by shifting the paradigm to serving the computational power as processes instead of web service threads, therefore customers can gain more flexible, configurable controls over their programs on the cloud.

This approach does reveal less security problems when the customer programs and data are appropriately encapsulated in process containers and governed accordingly by provider's hosts. The built-in encryption mechanism in the process containers conveniently hides the content. The access control policy is cryptographically enforced. Moreover, the process container carries the cloud provider's permissions together with the content and enforces it for each running program. As a result, the offered paradigm protects both the customer's and the provider's interests.

The proposed security paradigm covers a wide domain of PaaS security. The paper gives a framework that permits to obtain practical and open solutions to PaaS security problems. An experimental testbed has been developed for the proposed PaaS cloud. The initial experiments on this testbed indicated that the paradigm is convenient as well as open to development. Experiments and research are going on, including improved implementations. Further discussions, formalizations and analysis of the paradigm must be conducted for better understanding of its security behavior.

REFERENCES

1. *Apache Stratos*, https://stratos.apache.org/, accessed: 10.05.2015.
2. *AWS Elastic Beanstalk*, https://aws.amazon.com/elasticbeanstalk/, accessed: 10.05.2015.
3. *Google App Engine*, https://cloud.google.com/appengine/, accessed: 10.05.2015.
4. *Heroku*, https://www.heroku.com/, accessed: 10.05.2015.
5. A. S. TANENBAUM, *Modern Operating Systems*, Pearson Education International, 2009.
6. L. RODERO-MERINO, L. M. VAQUERO, E. CARON, A. MURESAN, F. DESPREZ, *Building Safe PaaS Clouds: A Survey on Security in Multitenant Software Platforms*, Computers & Security, **31**, *1*, pp. 96-108, 2012.
7. M. T. SANDIKKAYA, B. ODEVCI, T. OVATMAN, *Practical Runtime Security Mechanisms for an aPaaS Cloud*, IEEE Global Communications Conference (Globecom), pp. 53-58, 2014.
8. L. IBRAIMI, M. ASIM, M. PETKOVIC, *Secure Management of Personal Health Records by Applying Attribute-Based Encryption*, 2009 6th International Workshop on Wearable Micro and Nano Technologies for Personalized Health (pHealth), pp. 71-74, 2009.
9. M. T. SANDIKKAYA, A. HARMANCI, *Security Problems of Platform-as-a-Service (PaaS) Clouds and Practical Solutions to the Problems*, 2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS), pp. 463-468, 2012.
10. G. CZAJKOWSKI, L. DAYNES, *Multitasking Without Comprimise: A Virtual Machine Evolution*, Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 125-138, 2001.
11. G. BACK, W. C. HSIEH, *The KaffeOS Java Runtime System*, ACM Transactions on Programming Languages and Systems, **27**, *4*, pp. 583-630, 2005.
12. K. SUN, Y. LI, M. HOGSTROM, Y. CHEN, *Sizing Multi-space in Heap for Application Isolation*, Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, pp. 647-648, 2006.
13. N. GEOFFRAY, G. THOMAS, G. MULLER, P. PARREND, S. FRENOT, B. FOLLIOT, *I-JVM: a Java Virtual Machine for Component Isolation in OSGi*, IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09), pp. 544-553, 2009.
14. JSR 121: *Application Isolation API Specification*, https://jcp.org/en/jsr/detail?id=121, accessed: 01.04.2015.

15. T. LINDHOLM, F. YELLIN, G. BRACHA, A. BUCKLEY, *The Java Virtual Machine Specification*, Java SE 8 Edition, Oracle Corporation, 2010.
16. *Trusted Computer System Evaluation Criteria*, U.S. Department of Defense, 1985.
17. M. CASASSA MONT, S. PEARSON, P. BRAMHALL, *Towards Accountable Management of Identity and Privacy: Sticky Policies and Enforceable Tracing Services*, Proceedings of 14th International Workshop on Database and Expert Systems Applications, pp. 377-382, 2003.
18. J. GOSLING, B. JOY, G. STEELE, G. BRACHA, A. BUCKLEY, *The Java Language Specification*, Java SE 8 Edition, Oracle Corporation, 2010.
19. J. BETHENCOURT, A. SAHAI, B. WATERS, *Ciphertext-Policy Attribute-Based Encryption*, IEEE Symposium on Security and Privacy SP'07, pp. 321-334, 2007.
20. P. PAILLIER, *Public-key Cryptosystems Based on Composite Degree Residuosity Classes*, Advances in Cryptology–EUROCRYPT'99, pp. 223-238, 1999.
21. T. ELGAMAL, *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, Advances in Cryptology, pp. 10-18, 1985.
22. C. GENTRY, *Fully Homomorphic Encryption Using Ideal Lattices*, Proceedings of the 41st annual ACM Symposium on Theory of Computing – STOC'09, pp. 169-178, 2009.
23. C. GENTRY, S. HALEVI, N. P. SMART, *Homomorphic Evaluation of the AES Circuit*, Advances in Cryptology – CRYPTO 2012, pp. 850-867, 2012.
24. C. W. FLETCHER, M. VAN DIJK, S. DEVADAS, *Towards an Interpreter for Efficient Encrypted Computation*, Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop, pp. 83-94, 2012.
25. M. BRENNER, J. WIEBELITZ, G. VON VOIGT, M. SMITH, *Secret Program Execution in the Cloud Applying Homomorphic Encryption*, 2011 Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference (DEST), pp. 114-119, 2011.
26. D. ZHURAVLEV, I. SAMOILOVYCH, I. BONDARENKO, Y. LAVRENYUK, *Encrypted Program Execution*, 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 817-822, 2014.
27. S. GARG, C. GENTRY, S. HALEVI, M. RAYKOVA, A. SAHAI, B. WATERS, *Candidate Indistinguishability Obfuscation and Functional Encryption for All Circuits*, IEEE 54th Annual Symposium on Foundations of Computer Science, pp. 40-49, 2013.