Towards PaaS Offering of BPMN 2.0 Engines: A Proposal for Service-Level Tenant Isolation

Majid Makki(✉), Dimitri Van Landuyt, and Wouter Joosen

imec-DistriNet, KU Leuven, 3001 Heverlee, Belgium
{majid.makki,dimitri.vanlanduyt,wouter.joosen}@cs.kuleuven.be

Abstract. Business processes modeling and management solutions provide powerful abstraction mechanisms for the control flow of complex, task-driven applications, and as such allow for better alignment with business-related concerns. Despite the existence and wide adoption of standardized business process management languages such as WS-BPEL and BPMN 2.0, workflow engines in current Platform-as-a-Service (PaaS) offerings are in practice more restricted, in part for reasons such as vendor lock-in, but also due to restrictions of multi-tenant environments.

In this paper, we explore the main security-related problems caused by offering BPMN2-compliant workflow engines in a multi-tenant PaaS environment, particularly focusing on threats caused by misbehaving tenants and the lack of proper tenant isolation. In addition, we propose a service-level tenant isolation framework that allows PaaS offerings to support workflow engines which comply with the BPMN 2.0 standard, and we discuss the technical feasibility of implementing this framework using Java technologies such as OSGi and the Resource Consumption Management API (JSR-284).

Keywords: Platform-as-a-Service (PaaS) · Workflow engines Multi-tenancy · Untrusted code · Tenant isolation

1 Introduction

Platform-as-a-Service (PaaS) is a category of cloud computing services where the execution platform is offered to software teams for facilitating the development, deployment and maintenance of applications [1,2]. When optimized resource utilization is among the main goals, different applications may share a single installation of the execution platform in a multi-tenant fashion. Workflow engines, in charge of executing business processes, can be part of a PaaS offering and, thus, shared among multiple tenant applications.

Renowned workflow engines in PaaS offerings, such as Amazon SWF [3] and Fantasm in Google App Engine [4], incur a high degree of vendor lock-in and are limited in functionality and suboptimal vis-à-vis utilization of resources. Since these engines have their own custom (i.e. non-standard) workflow modeling languages, the application will be, *de facto*, locked-in by the PaaS provider due to

© Springer International Publishing AG, part of Springer Nature 2018
Z. Á. Mann and V. Stolz (Eds.): ESOCC 2017, CCIS 824, pp. 5–19, 2018.
https://doi.org/10.1007/978-3-319-79090-9_1

high cost of porting (cf. [5]). In addition, some features, such as human tasks or advanced event handling mechanisms which are commonly used in state-of-the-art business process automation (cf. [6]), are not supported out of the box. Supporting such features requires quite some ad-hoc engineering effort by the application developers. Furthermore, despite sharing the execution environment of the workflow engine between multiple tenant applications, these solutions require separate environments for executing workflow tasks of each distinct tenant application. The latter decreases resource efficiency whose maximization is a principal goal of cloud computing [7].

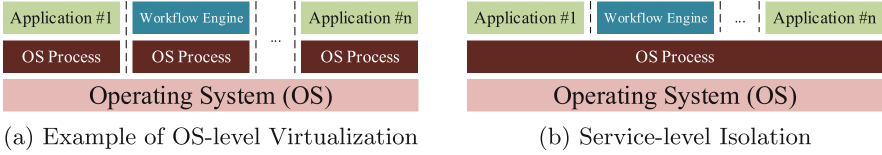
These problems can be solved by offering workflow engines that comply with the Business Process Modeling and Notation 2.0 (BPMN 2.0) [8] specification which, next to the Business Process Execution Language (BPEL) [9], is the standard increasingly being adopted in practice. The standardized nature of such engines increases the portability of applications developed using them. In addition, thanks to accumulated experience of decades which is behind BPMN 2.0, these engines do not lack necessary and mainstream functional features. Furthermore, these engines are capable of executing workflow tasks in the same execution environment as the engine itself. Thanks to this capability, resources are utilized more efficiently.

However, PaaS offering of BPMN2-compliant engines causes certain security threats which necessitates specific protection measures well beforehand. The principal source of threats is the untrusted tenant-provided code of workflow tasks that will be executed in an execution environment shared between the PaaS provider and multiple, possibly competing, tenants (cf. [10]). For instance, conflicting access to IO resources is possible. As an alternative example, tenants may exhaustively consume resources such as memory and bring down the service entirely.

The state-of-the-art protection mechanism against such threats is OS-level virtualization, i.e. hypervisors [11] or containers [12], where granularity-level of tenant isolation is, as shown in Fig. 1(a), that of Operating System (OS) processes. This requires having at least one active OS process for each tenant which implies that quite some resources are reserved even if the tenant application does not impose any load. Moreover, OS-level virtualization is not sufficient for all functionalities of BPMN2-compliant engines because they execute some workflow tasks within the same OS process as the engine itself[1] which requires sharing a single OS process between the PaaS provider and tenants. For more efficient utilization of resources and being compatible with the nature of BPMN2-compliant engines, tenant isolation has to take place at a higher level in the computational stack. As depicted by Fig. 1(b), this is the level where the service itself is implemented.

This work-in-progress paper proposes a *service-level* tenant isolation framework for enabling PaaS offering of BPMN 2.0 engines based on Java technologies. We formulate a concrete research problem by analyzing the BPMN 2.0 specification and widely-used BPMN2-compliant engines. Furthermore, given the fact that the *absolute* majority of BPMN2-compliant engines are Java-based [13],

[1] This is required by the BPMN 2.0 specification for some types of tasks.



(a) Example of OS-level Virtualization (b) Service-level Isolation

Fig. 1. OS-level virtualization requires having separate OS processes for each tenant application and the workflow engine while service-level isolation allows running all code inside a single OS process.

we present an initial outline of a solution based on Java-related technologies. The proposed solution takes threads as units of isolation for executing untrusted code of tenant applications to overcome the aforementioned insufficiency of the OS-level virtualization approach and the suboptimal resource utilization thereof. The technical feasibility of the solution is shown by explaining how existing technologies, such as OSGi [14] and the Java Resource Consumption Management API (JSR-284 [15]), enable its implementation.

The rest of this paper is structured as follows. Section 2 analyzes the research problem. Section 3 presents the solution outline along with remarks on its technical feasibility. Section 4 briefly contrasts this proposal with related work. Finally, Sect. 5 concludes the paper.

2 Problem Statement

This section analyzes the most compelling security threats caused by the PaaS offering of BPMN 2.0 engines and formulates the research problem as a number of concrete requirements.

2.1 Security Threat Analysis

We have systematically analyzed and prioritized the security threats using the **STRIDE**[2] threat model [16,17]. The most problematic security threats, insofar as this paper is concerned, are related to two *core* features of BPMN 2.0 namely `Script Task` and `Service Task` activity types. The former is required by the standard to be "executed by a business process engine" [8]. This implies that the same OS process running the engine is responsible for executing the `Script Task`. While the standard does not require `Service Task` activities to be executed by the same Operating System (OS) process running the engine, most well-known and enterprise-ready BPMN 2.0 engines, such as jBPM [18] and Activiti [19], allow defining `Service Task` activities that are executed within the same OS process running the engine. Retaining this additional feature is

[2] The acronym stands for six threat categories namely **S**poofing, **T**ampering with Data, **R**epudiation, **I**nformation Disclosure, **D**enial of Service and **E**levation of Privilege.

essential for avoiding the overhead of remote service invocation, e.g. (de-)serialization and network delay.

In a multi-tenant context of a PaaS offering, the code of `Script Task` and `Service Task` activities belong to untrusted tenant applications using the engine. Executing untrusted code of tenants in the same OS process running the engine incurs different types of security threats (cf. [10,20,21]). Tampering with Data, Information Disclosure, Denial of Service and Elevation of Privilege are identified as the most important threat categories in this specific context and are discussed below.

Tampering with Data. Since tenant-provided code may access IO resources, one tenant application may modify another tenant's data stored on a shared device. In addition, a tenant application may modify the value of in-memory object references belonging to or shared with other tenants.

Information Disclosure. A tenant application may read another tenant's data by, e.g., listening on a network port belonging to the other tenant. Similarly, a tenant application may access in-memory object references belonging to or shared with other tenants.

Denial of Service. One tenant application may disrupt the PaaS offering entirely either by *using up* computational resources or by misusing part of the API shared among all tenants. The resources of concern are CPU cycles, memory space and IO bandwidth (both storage and network). Misuse of shared API can be either in form of killing the OS process hosting the service or in form of locking shared objects *indefinitely*.

Elevation of Privilege. By creating a thread which is not under control of the framework, one tenant application may increase its privileges and act without constraints imposed by the framework.

2.2 Requirements

Since these threats are caused by code running within a single OS process, protection against them has to take place inside that OS process as well. Therefore, a *service-level* tenant isolation framework is needed which fulfills the following functional requirements:

- FR1: The framework should guarantee that no tenant application may access objects or primitive values belonging to other tenants.
- FR2: The framework has to guarantee that shared system objects and references accessible for tenant applications can neither be locked indefinitely nor be modified by any of them.
- FR3: The creation of threads has to be entirely mediated by the framework.
- FR4: Permission of killing the OS process has to be denied for all tenant applications.
- FR5: The framework should check tenant permissions before granting access to any IO resource (e.g. a file on storage device or a network port).

- FR6: An upper limit has to be put on CPU usage, memory consumption and IO bandwidth (both storage and network) on a per-tenant basis.
- FR7: Upper limit imposition on resource consumption has to be so flexible that resource utilization maximizes. In other words, if there are unused resources that can be allocated safely, the framework should let tenant code exceed the limits to some extent.

In addition, fulfillment of the above functional requirements should respect the following quality requirements:

- QR1: All tenant isolation measurements should be enforced transparently by the framework. In other words, code of tenant applications has to be entirely decoupled from the tenant isolation framework.[3]
- QR2: The relative performance overhead of the framework compared to OS-level virtualization tactics (cf. [11,12]) is required to be in an acceptable margin.

3 Service-Level Tenant Isolation

This section outlines a service-level tenant isolation framework as a solution for fulfilling the above requirements and shows technical feasibility of the framework. Section 3.1 presents the framework architecture conceptually. Section 3.2 elaborates on partial fulfillment of FR1 while Sect. 3.3 supplements it and discusses static code restriction which is required for fulfillment of FR2 and FR3. Supplementary measures for fulfillment of FR2 and FR3 are presented in Sect. 3.4 while Sect. 3.5 deals with permission checking mechanism which is required for FR2, FR3, FR4 and FR5. Finally, Sect. 3.6 explains how the framework realizes FR6 and FR7. The qualitative requirements are taken into account orthogonally throughout this section.

3.1 Overall Architecture

Figure 2 shows the principal components running within a single OS process in three layers: (i) the bottom layer is the Java execution platform and the components it provides, (ii) the middle layer is where the service components of the PaaS offering, including components of the tenant isolation framework, sit, and (iii) the top layer consists of code of tenant applications built upon the PaaS offering.

The tenant isolation framework consists of seven main components which are introduced gradually throughout this section. The two front-end components of the framework which are directly used by tenants are `Deployment Service` and `Workflow Execution Service`. The former is responsible for deploying tenant applications into the PaaS environment. The latter is responsible for starting new workflow executions or continue/monitor/abort existing ones by running the code of tenant applications in isolation from other tenants.

[3] This is required for portability of tenant applications to other instances of the same BPMN 2.0 engine where the tenant isolation framework is not used.

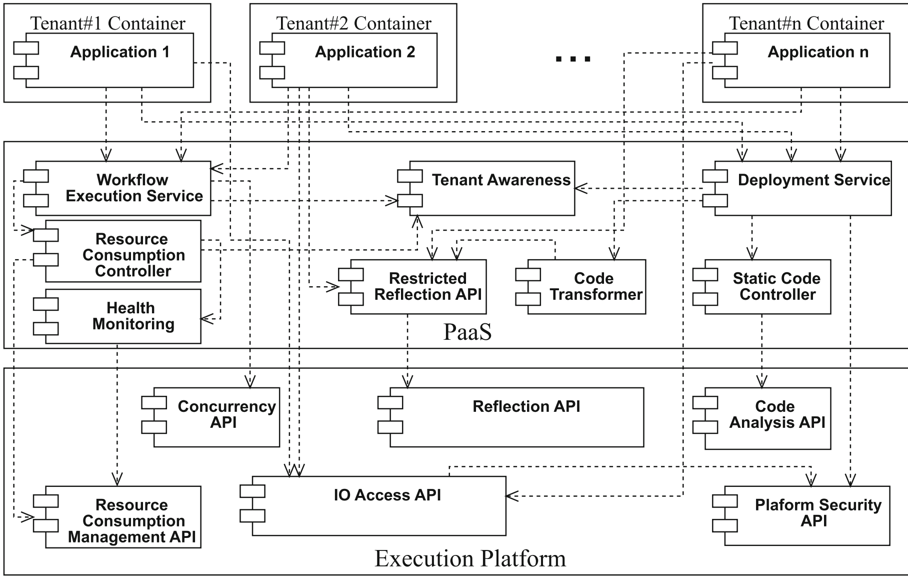


Fig. 2. Building blocks of the framework in relation to tenant applications and the execution platform

For imposing isolation measures, the `Workflow Execution Service`, using an existing BPMN2-compliant engine such as jBPM [18], runs the untrusted code of tenant applications in separate threads and guarantees that each active thread is associated with only one tenant application at a time. Once each thread finishes its job, it can be reused for other tenant applications using a thread pool provided by the Java `Concurrency API`. The `Workflow Execution Service` leverages upon the `Concurrency API` of the Java execution platform for handling threads. Furthermore, it uses the `Tenant Awareness` component of the framework which is responsible for keeping track of associations between threads and tenants. Thus, the main cornerstone of the proposed solution is taking threads as units of tenant isolation just as OS-level virtualization tactics take OS processes as units of tenant isolation.

3.2 Tenant Containers

In order to dedicate a separate referencing space for objects of each tenant application (cf. FR1), the `Deployment Service` deploys each application in a distinct tenant container. As opposed to containers of OS-level virtualization approach, containers shown in Fig. 2 are managed inside a single OS process. OSGi [14] bundles provide exactly this containerization functionality. OSGi loads each bundle using a distinct Java classloader and sets the bootstrap classloader, which is responsible for loading core Java classes, as the parent of each bundle

classloader. Hence, the code of each bundle can access fields of its own classes and *static* fields of core Java classes.

By containing the code of each tenant application in a separate OSGi bundle, the FR1 requirement will be partially fulfilled. For complete fulfillment of FR1, cross-bundle communication between tenant application bundles has to be forbidden which is the topic of next section along with realization of FR2 and FR3.

3.3 Static Code Restriction

Access of tenant applications to other classes and interfaces has to be restricted for fulfillment of FR1, FR2 and FR3. API restriction is required both statically and dynamically. Static restriction takes place only once at deployment time by the `Static Code Controller` (cf. Fig. 2). This is done in multiple stages using code analysis facilities provided by the Java platform and tools built upon it.

Cross-bundle Communication. Each OSGi bundle declares the list of classes it imports from other bundles. This is used for cross-bundle communication. By imposing limits on this list, the `Static Code Controller` guarantees that no cross-bundle communication is possible between two tenant applications and, thus, completes the realization of FR1.

Blacklist. One of the main elements of `Static Code Controller` is `BlacklistService` which maintains a list of classes, methods and fields that tenant applications are not allowed to use.

Given the structure of classloaders in OSGi, security vulnerabilities pertaining to shared object locks and modifications (cf. FR2) are caused by four Java programming constructs related to classes loaded by the bootstrap classloader: (i) *static* field declarations, (ii) reference updates on *static* fields, (iii) changing state of objects referenced by *static* fields, (iv) *static synchronized* methods, and (v) *synchronized* blocks locking *static* fields [10,20,21]. The `BlacklistService` searches for occurrences of these constructs in all classes loaded by the bootstrap classloader using the Java source code querying facilities provided by the Spoon library [22] as well as call graph construction and reference analysis (a.k.a. points-to analysis) mechanisms of the Soot framework [23]. A call graph consists of nodes and edges representing Java methods and invocation relationship between them respectively. Reference analysis helps resolving non-static access to objects referenced by *static* fields of concern.

In addition, to further comply with FR3, the `BlacklistService` adds the `java.lang.Thread` class to the blacklist. Furthermore, using the Spoon library, it adds any method in the `java.util.concurrency` package capable of instantiating new threads or intervening in the life-cycle of an existing thread.

Acceptance Policy. The final stage of `Static Code Controller` involves accepting or rejecting the tenant application code. It checks whether the code of a tenant application directly or indirectly deals with blacklisted constructs.

```
SecurityManager sm = System.getSecurityManager();
if (BlacklistService.isBlacklisted(this))    {
    if (sm != null) {
        sm.checkPermission(new
            ReflectPermission("evadeBlacklist"));
    }
}
super.originalMethod(...); // pseudocode
```

Listing 1.1. Restricting access of tenant applications to Java Reflection API.

For instance, it checks whether a blacklisted `static synchronized` is invoked or the constructor of the `Thread` class is used. The first step for doing so is creating a call graph using the Soot framework. Afterwards, the call graph has to be traversed to see if any of the fields, methods or classes in the blacklist is used by the methods included in the call graph. Furthermore, using the reference analysis mechanism provided by Soot, it has to be verified whether objects referenced by *static* fields in the blacklist are modified indirectly (e.g. by means of an intermediate local reference).

3.4 Dynamic Code Restriction

Restricting code of tenant applications statically is not sufficient because all the malicious operations, which can be detected statically, can be done dynamically as well using the `Reflection API`. Therefore, `Restricted Reflection API` should be used by tenant applications instead of the original Java `Reflection API` (cf. Fig. 2). This, however, can reintroduce the vendor lock-in problem that QR1 requires to avoid. Therefore, tenant applications are allowed to use the original `Reflection API` but at deployment-time, the `Deployment Service` asks the `Code Transformer` to transform tenant code such that the original `Reflection API` is replaced by `Restricted Reflection API`. This is feasible and straightforward because the `Restricted Reflection API` has exactly the same package structure and exposes exactly the same API as the original one but with different behavior in some cases. The `Code Transformer` component employs the transformation utilities provided by the Spoon framework [22].

The behavior difference is summarized by Listing 1.1 where `this` either refers to a `Field` object whose `get` method is called, a `Constructor` object whose `newInstance` method is invoked or a `Method` object whose `invoke` method is called. Determining whether the use of these class members are blacklisted for tenant applications, the same `BlacklistService`, which maintains the blacklist, is used. Since the blacklist is prepared once in the entire life-time of the application and kept in memory for subsequent uses, this does not impose a significant performance overhead (cf. QR2).

As shown, Java `SecurityManager` is used to check whether `evadeBlacklist` permission is granted to the OSGi bundle requesting the reflective operation. This permission has to be granted only to trusted bundles, i.e. not to bundles containing code of tenant applications. The `checkPermission` method throws

an `AccessControlException` if the required permission is not granted to the bundle requesting the reflective operation.

The next section elaborates on how the permission checking mechanism of the Java `SecurityManager` works and on how it is employed by the tenant isolation framework.

3.5 Permission Checks

In addition to the above permission checking, the tenant isolation framework enforces permission checks on invocations of `System.exit` method (cf. FR4) and on every IO access (cf. FR5). Permission checks are automatically done by the Java platform itself once the `SecurityManager` is enabled. The role of the framework, hence, is limited to enabling the `SecurityManager` and granting permissions properly.

The `SecurityManager` relies on a mechanism called *stackwalking* and associates a permission set to each *protection domain* (cf. [24,25]). In OSGi, there exists one protection domain for each bundle. The set of permissions of each tenant application is granted to it by the `Deployment Service` at deployment time. Tenant permission sets are, in principle, a combination of `FilePermission` and `SocketPermission` for restricting their access to IO resources which is required by FR5. By denying the `RuntimePermission("exitVM")` to tenant bundles, FR4 is also fulfilled. All other bundles, which do not contain tenant-provided code, are granted `AllPermission`.

When permission p is required, the Java platform `SecurityManager` triggers stackwalking, i.e. tracing the entire method invocation stack which has led to the point of permission checking, and verifies that

$$\forall_{pd \in PD_s} p \in P_{pd} \quad (1)$$

where PD_s is the set of all protection domains involved in the scanned method invocation stack s and P_{pd} is the set of permissions granted to the protection domain pd. This way, the permissions of tenant application for whom permission p has to be checked are taken into account and retrieved from its corresponding protection domain.

Permission checks are not required when the running code is trusted, e.g. when PaaS management and monitoring components are executed. In order to avoid the performance overhead of the `SecurityManager` when it is not needed, the framework enables it only when tenant application code is executed and disables it otherwise. The `TenantWorkflowExecutor`, which is responsible for starting/resuming/aborting a tenant workflow, toggles the `ToggleableSecurityManager` shown in Fig. 3 before and after workflow execution using `enable` and `disable` methods of the latter. These methods change the value of a `ThreadLocal` variable which is used in the `checkPermission` method according to Listing 1.2.

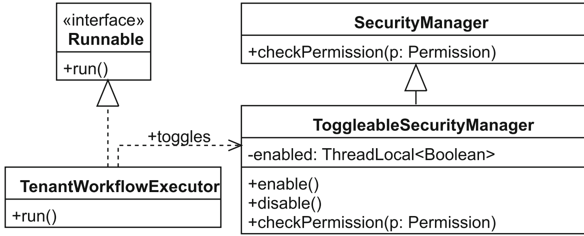


Fig. 3. The framework extends the `SecurityManager` such that it can be enabled only when tenant-provided code is executed. This is done by means of a `ThreadLocal` variable toggled before and after a tenant workflow executes.

```
if (enabled.get()) {
    super.checkPermission(p);
}
```

Listing 1.2. Evading permission check when it is not required.

3.6 Resource Consumption Control

Fulfilling FR6 requires associating a service-level agreement (SLA) to each tenant application. Listing 1.3 shows the structure of a tenant SLA. The framework uses the Java Resource Consumption Management API (JSR-284) [15] for imposing limits on resource consumption of each tenant application. On a per-tenant basis, `ResourceMeter` instances are created for each element of the SLA. Meters are notified by the Java platform every time new information about their corresponding resources allocation/release is available. The `Resource Consumption Controller` creates tenant meters only once (the first time they are needed) by consulting the `Tenant Awareness` component which has access to SLAs. Once meters are created, they will be associated with a tenant-specific `ResourceContext`. Before executing the untrusted code of any tenant, the `TenantWorkflowExecutor` associates the tenant meters to the executing thread according to Listing 1.4.

The meters provided by the Java platform are themselves capable of imposing a limit on consumption of IO-related resources. Hence, it is sufficient to choose the right type of meter for each resource type. The `BoundedMeter` is used for `maxOpenFiles`, `maxOpenSockets` and `maxOpenDatagrams` and the `ThrottledMeter` is used for other IO-related resources. Both of these meters are capable of imposing a limit on resource usage. The only difference is that the former is appropriate for cases dealing with absolute numbers (e.g. number of open files) while the latter best suits cases where a rate is involved (e.g. bytes read from file system per seconds).

The meters provided by the Java platform however are not capable of imposing a limit on memory and CPU usage. The best they can do is to notify when the limit is reached. Our framework employs Quasar Fibers [26] instead of original

```java
public class TenantSLA {

    private long maxCpuUsage; // CPU nanoseconds per second
    private long maxMemoryUsage; // bytes

    private int maxOpenFiles;
    private long maxReadDiskRate; // bytes per second
    private long maxWriteDiskRate; // bytes per second

    private int maxOpenSockets; // TCP sockets
    private long maxReadSocketRate; // bytes per second
    private long maxWriteSocketRate; // bytes per second

    private int maxOpenDatagrams; // UDP datagrams
    private long maxReadDatagramRate; // bytes per second
    private long maxWriteDatagramRate; // bytes per second

    // getters and setters

}
```

Listing 1.3. Structure of Tenant SLA.

```java
ResourceContextFactory factory = ResourceContextFactory.getInstance();
ResourceContext rc = factory.lookup(tenantId);
rc.bindThreadContext(); // binds to the current thread
// workflow execution code
rc.unbindThreadContext(); // unbinds from the current thread
```

Listing 1.4. Binding tenant `ResourceContext` to threads before starting workflow execution.

Java threads in order to safely suspend the untrusted code of tenants when they consume too much memory space or CPU time. Since Fiber suspension is done at the level of the JVM rather than that of the OS kernel, regularly suspending them does not impose much overhead. Furthermore, tenant-specific information about consumption level can be used for resuming suspended Fibers whereas execution of original Java threads are left to the OS kernel which does not have any tenant-specific information.

Fiber suspension may take place after specific checkpoints. The memory usage is checked every time a new object is instantiated. Therefore, the bytecode of the `Object` constructor is manipulated to enforce a memory limit checkpoint. Once the responsible `NotifyingMeter` notifies the surpass of memory limit by a specific tenant, a tenant-specific boolean variable will be modified to indicate that the corresponding tenant cannot create new objects anymore. The boolean variable will be consulted by the memory checkpoint inside of the Object constructor.

The CPU checkpoints, however, are more widespread. They have to guarantee that tenants cannot evade CPU usage control mechanism. Hence, two types of checkpoints are inserted by bytecode manipulation of both the Java API and the untrusted code of tenant: (i) inside every loop structure and (ii) inside every recursion (be it direct or indirect). Manipulating the Java API is required

because tenants may exploit it by means of method arguments. Similarly, a boolean variable is checked in every checkpoint and if the tenant code has to be suspended due to excess of CPU usage, it will be suspended. Despite the widespread nature of CPU checkpoints, the performance overhead is not expected to be high because under normal circumstances every checkpoint will amount to a simple if statement involving a boolean variable.

FR7 requires managing some types of resources flexibly. Flexible resource consumption means allowing tenants surpassing the limits defined in their SLAs when there are sufficient amount of resources available for allocation. This involves too much risk in case of memory usage, number of open files and number of open sockets because once tenants go beyond their limits on these types of resources, there is no guarantee that the system can push them back to their borders (cf. [10,20,21] for the case of memory usage). However, that risk is not relevant in case CPU usage and IO bandwidth because their consumption is time-dependent by nature and thus can be reclaimed by the system if need be.

The following condition determines whether resource allocation can be done regardless of the fact that a tenant SLA limit is reached:

$$total_r + amount_r \leq l_r \times capacity_r \tag{2}$$

where r is a resource of concern, $total_r$ is the total consumption amount of all tenants before approving the new request, $amount_r$ is the requested amount, $capacity_r$ is the total capacity of the system on r and l_r is the leniency factor between 0 and 1. When leniency factor is set to zero, every tenant will be strictly restricted to its SLA regardless of resource availability, i.e. unallocated amounts. When it is set to one, the unallocated amounts will be used for letting more active tenants going beyond their SLA limits.

The `System Health Monitoring` is consulted component for retrieving the total usage. Capacity is a set by system administrators while total usage is retrieved from the `ResourceContextFactory` provided by the Java platform. The latter does not calculate the total usage every time queried. Instead, it keeps track of total amounts on every resource allocation and release. Decisions of this approver are safe because they are made about resources measured on a per-second basis and surpassing their limits will have no effect beyond the measurement window which is two seconds according to the API. In other words, it is guaranteed that these resources can be throttled back to the limits defined in tenant SLAs once the load on the system increases.

4 Related Work

In this section, we briefly compare this paper with related work.

PaaS Offering of Workflow Engines. Pathirage et al. [27] proposes an architecture for PaaS offering of Apache ODE [28] which is a workflow engine complying with WS-BPEL. Since all activities involving code execution are remote web-services, the security threats that we covered in this paper are not relevant

for that work. However, delegating execution of all untrusted activity code to remote web-services both reduces efficiency of resource utilization and imposes the overhead of remote invocation (e.g. network delay and serialization). Yu et al. [29] claims having enabled jBPM [18], a BPMN2-compliant engine, to be offered as a PaaS. However, the security threats discussed in this paper are overlooked altogether. Amazon SWF [3] and Fantasm on Google App Engine [4] are production ready PaaS offerings of workflow engines. However, applications developed using them highly suffer from vendor lock-in problem in terms of both code and data. Furthermore, resource utilization is sub-optimal due to adoption of OS-level virtualization tactics.

OS-level Virtualization. Similar isolation measures can be imposed by means of containers (cf. [12]) or virtual machines (cf. [11]). However, in case of Java, a separate instance of the JVM should be started for each tenant which reduces efficiency of resource utilization. Furthermore, these solutions are totally insufficient for offering BPMN2-compliant engines as the latter runs `Script Task` and, in some cases, `Service Task` activities in the same OS process as the workflow engine itself.

Java Language Vulnerabilities. Security threats related to running untrusted code in Java threads are discussed in [10,20,21]. These problems are caused by the shared nature of *static* fields, blocking effect of *static synchronized* methods, reference leaks and shared nature of computational resources. Our threat analysis is based on these works and we have proposed ideas for solving some of them and workarounds for some others based on existing technologies.

Application Performance Isolation. There are a number of works dealing with performance isolation for Software-as-a-Service (SaaS) applications [30–33]. While these works deal with SLAs expressed in external properties of an application such as response-time and throughput, the SLAs are defined in system-level terms such as CPU usage. This is because these works do not deal with the issues of running untrusted code in a shared execution environment. Krebs et al. have proposed a framework for determining resource usage based on the aforementioned external properties [34]. However, their solution requires categorization of all possible requests into different groups beforehand which is not feasible in case of PaaS where application requests are not known by the PaaS provider.

5 Conclusion

Business process automation is a common practice supported by a set of mature standards (e.g. BPMN 2.0 and WS-BPEL) and numerous workflow engines that implement these standards. Due to the specific deployment model of multi-tenancy in a Platform-as-a-Service (PaaS) context, full support of these standards requires additional attention to security threats caused by misbehaving tenants. We have presented an outline of a framework for tenant isolation in the context of co-existing business processes of different tenants, and we have discussed its practical feasibility for the Java environment.

Advancing this work fits into our ongoing research on the key trade-offs related to multi-tenancy between resource optimization, customization support (e.g. by means of tenant-provided tasks), security (tenant isolation) and portability of business processes across different cloud providers. We have implemented the permission checking and resource consumption mechanisms of the framework. In follow-up work, we will further implement the code restriction part and evaluate the proposed framework vis-à-vis performance overhead compared to the OS-level virtualization approach and dimension of tenant code restrictions (i.e. determining how limited tenant applications will be in using the Java API given the restrictions imposed by the framework).

Acknowledgement. This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS), the strategic basic research (SBO) project DeCoMAdS, and the MuDCads O&O project.

References

1. Rimal, B.P., Choi, E., Lumb, I.: A taxonomy and survey of cloud computing systems. In: INC, IMS and IDC, pp. 44–51 (2009)
2. Walraven, S., Truyen, E., Joosen, W.: Comparing paas offerings in light of SaaS development. Computing **96**(8), 669–724 (2014)
3. AWS: Amazon Simple Workflow Service (Amazon SWF). https://aws.amazon.com/documentation/swf/. Accessed 12 June 2017
4. Google: Google App Engine Fantasm. https://cloud.google.com/appengine/articles/fantasm. Accessed 12 June 2017
5. Opara-Martins, J., Sahandi, R., Tian, F.: Critical review of vendor lock-in and its impact on adoption of cloud computing. In: 2014 International Conference on Information Society (i-Society), pp. 92–97. IEEE (2014)
6. Ko, R.K., Lee, S.S., Wah Lee, E.: Business process management (BPM) standards: a survey. Bus. Process Manag. J. **15**(5), 744–791 (2009)
7. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
8. OMG: Business Process Model and Notation 2.0. http://www.omg.org/spec/BPMN/2.0/PDF/. Accessed 04 Aug 2015
9. OASIS: Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Accessed 04 June 2016
10. Rodero-Merino, L., Vaquero, L.M., Caron, E., Muresan, A., Desprez, F.: Building safe PaaS clouds: a survey on security in multitenant software platforms. Comput. Secur. **31**(1), 96–108 (2012)
11. Li, Y., Li, W., Jiang, C.: A survey of virtual machine system: current technology and future trends. In: 2010 Third International Symposium on Electronic Commerce and Security (ISECS), pp. 332–336. IEEE (2010)
12. Bernstein, D.: Containers and cloud: from LXC to docker to kubernetes. IEEE Cloud Comput. **1**(3), 81–84 (2014)
13. Wikipedia: List of BPMN Engines. https://en.wikipedia.org/wiki/List_of_BPMN_2.0_engines. Accessed 05 July 2017
14. OSGi-Alliance: OSGi specification (2012). https://osgi.org/download/r4v43/osgi.core-4.3.0.pdf. Accessed 19 April 2017

15. JCP: JSR 284: Resource Consumption Management API. https://jcp.org/en/jsr/detail?id=284. Accessed 12 June 2017
16. Microsoft: The stride threat model (2015). https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx. Accessed 19 April 2017
17. Shostack, A.: Threat Modeling: Designing for Security. Wiley, New York (2014)
18. RedHat-JBoss: jBPM. http://www.jbpm.org/. Accessed 04 June 2017
19. Alfresco: Activiti User Guide. https://www.activiti.org/userguide/. Accessed 24 May 2017
20. Czajkowski, G., Daynés, L.: Multitasking without comprimise: a virtual machine evolution. ACM SIGPLAN Not. **36**, 125–138 (2001)
21. Herzog, A., Shahmehri, N.: Problems running untrusted services as Java threads. Certification Secur. Inter-Organ. E-Serv. **177**, 19–32 (2004)
22. Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: a library for implementing analyses and transformations of Java source code. Softw. Pract. Exp. **46**(9), 1155–1179 (2016)
23. Lam, P., Bodden, E., Lhoták, O., Hendren, L.: The soot framework for Java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011), vol. 15, p. 35 (2011)
24. Oracle: Java 8 SE platform security. https://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html. Accessed 19 April 2017
25. Gong, L., Ellison, G.: Inside Java (TM) 2 Platform Security: Architecture, API Design, and Implementation. Pearson Education, London (2003)
26. Parallel Universe: Quasar. http://docs.paralleluniverse.co/quasar/. Accessed 09 July 2017
27. Pathirage, M., Perera, S., Kumara, I., Weerawarana, S.: A multi-tenant architecture for business process executions. In: 2011 IEEE International Conference on Web services (ICWS), pp. 121–128. IEEE (2011)
28. Apache: Apache ode. http://ode.apache.org/. Accessed 09 July 2017
29. Yu, D., Zhu, Q., Guo, D., Huang, B., Su, J.: jBPM4S: a multi-tenant extension of jBPM to support BPaaS. In: Bae, J., Suriadi, S., Wen, L. (eds.) AP-BPM 2015. LNBIP, vol. 219, pp. 43–56. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19509-4_4
30. Walraven, S., De Borger, W., Vanbrabant, B., Lagaisse, B., Van Landuyt, D., Joosen, W.: Adaptive performance isolation middleware for multi-tenant SaaS. In: 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), pp. 112–121. IEEE (2015)
31. Krebs, R., Loesch, M., Kounev, S.: Platform-as-a-service architecture for performance isolated multi-tenant applications. In: 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), pp. 914–921. IEEE (2014)
32. Krebs, R., Momm, C., Kounev, S.: Metrics and techniques for quantifying performance isolation in cloud environments. Sci. Comput. Program. **90**, 116–134 (2014)
33. Lin, H., Sun, K., Zhao, S., Han, Y.: Feedback-control-based performance regulation for multi-tenant applications. In: 2009 15th International Conference on Parallel and Distributed Systems (ICPADS), pp. 134–141. IEEE (2009)
34. Krebs, R., Spinner, S., Ahmed, N., Kounev, S.: Resource usage control in multi-tenant applications. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 122–131. IEEE (2014)