Security-by-Contract for the OSGi Platform

Olga Gadyatskaya, Fabio Massacci, Anton Philippov

Security-by-Contract for the OSGi platform

Olga Gadyatskaya, Fabio Massacci, and Anton Philippov

DISI, University of Trento, Italy,
{name.surname} @unitn.it

Abstract. The natural business model of OSGi is dynamic loading and removal of bundles or services on an OSGi platform. If bundles can come from different stakeholders, how do we make sure that one's services will only be invoked by the authorized bundles? A simple solution is to interweave functional and security logic within each bundle, but this decreases the benefits of using a common platform for service deployment and is a well-known source of errors. Our solution is to use the Security-by-Contract methodology (S×C) for loading time security verification to separate the security from the business logic while controlling access to applications. The basic idea is that each bundle has a *contract* embedded into its manifest, that contains details on functional requirements and permissions for access by other bundles on the platform. During bundle installation the contract is matched with the platform security policy (aggregating the contracts of the installed bundles). We illustrate the S×C methodology on a concrete case study for home gateways and discuss how it can help to overcome the OSGi security management shortcomings.

1 Introduction

The Open Services Gateway Initiative (OSGi) framework [1] is one of the most flexible solutions for the deployment of pervasive services in home, office, or automobile environments. OSGi-compatible implementations constitute the backbone of many recent proposals for embedded systems [2] or other industry-based services. The OSGi services are also the basic building blocks for service mash-ups extending the classical "smart homes" scenarios to richer settings [8].

The OSGi framework redefines the modular system of Java by introducing *bundles*: JAR files enhanced with specific metadata. The *service* layer connects bundles in a dynamic way with a publish-find-bind model for Java objects. An OSGi-based system has several advantages over the traditional JAR modules because it provides a robust integrated environment where bundles can be published and exported to be used by other bundles, handles bundle versioning for every new deployment and maintains the bundle lifecycles. Moreover, the bundles can be updated dynamically at run-time without restarting the system and seamlessly to other bundles.

As a result, an OSGi platform is expected to be highly dynamic. All pervasive and mash-up applications expect that bundles can be installed, updated or removed at any time depending on business needs, and also they can collaborate arbitrarily in order to ensure enhanced composite services.

From a security perspective, the possibility of bundle interactions is a threat for bundle owners. Since bundles can contain sensitive data or activate sensitive operations (such as locking doors of somebody's house), it is important to ensure that the security policy of each bundle owner is respected by other bundles. However, such aspects have been only partially investigated.

How do we make sure that one's services are invoked only by authorized bundles? A simple solution is to rely on service-to-service authentication to identify the services and then interleave functional and security logic into bundles, for example, by using aspect-oriented programming [11]. However, this decreases the benefits of using a common platform for service deployment and significantly hinders evolution and dynamicity: any change to the security policy would require redeployment of the bundle (even if its functionalities are unchanged). Vice versa, any changes in the bundle's code would require redeployment of security.

Our solution is to use the Security-by-Contract methodology [3, 5] for loading time security verification in order to separate security and the business logic while achieving a sufficient protection of bundles among themselves. S×C's basic idea is that each bundle will have a *contract* embedded into its manifest file. The contract contains details on the functional requirements and lists access permissions for other bundles on the platform. During installation of bundles the contract is extracted and matched with the platform security policy aggregating the contracts of all installed bundles. Thus, after the check we can be sure (under reasonable assumptions) that the incoming bundle respects the security requirements of other bundles.

Overall contribution of the paper is twofold. First, we improve the OSGi security by introducing a way for bundle providers to define their security policies and enforcing them on the platform. Second, we extend the S×C paradigm to a permission-based security system, opening a way for it to be adopted further for other platforms.

The rest of this paper is organized as follows. Section 2 introduces the S×C scheme for OSGi, Section 3 presents the smart home service gateway case study and discusses the security and functionality challenges, Section 4 presents the formal model of the OSGi system, the contract notation and the checks that the S×C framework performs to ensure security. Section 5 evaluates the proposed solution, Section 6 overviews the related work and Section 7 concludes the paper.

2 The S×C Architecture

We assume at least a high-level understanding of the main notions of the OSGi platform such as *bundles* (the OSGi components made by developers), *services* (plain old Java objects connecting bundles in a dynamic fashion by the means of publish-find-bind model), *lifecycle* API (the API to install, start, stop, update, and uninstall bundles) and *modules* – the layer that defines how a bundle can import and export code.

The S×C framework consists of two main components: the ClaimExtractor and the PolicyChecker. The verification workflow is described on Figure 1. Informally,

the S×C process starts when a new bundle B is loaded. The ClaimExtractor component then accesses the manifest file, retrieves the information about imported and exported packages and obtains the bundle contract. Then the ClaimExtractor reads the permissions.perm file, which contains local bundle permissions, extracts permissions requested by the bundle B and related to services retrieval, packages importing, requirements of bundles, etc., and combines this information into the overall "security claims and needs" of the bundle. Then the PolicyChecker component receives the result from the ClaimExtractor and matches it with the security policy of the platform, that aggregates the security policies of all the installed bundles, and with the functional state of the platform (installed bundles, running services, etc.). If the PolicyChecker failed on either of the checks, the bundle is removed from the platform. Otherwise, it is installed and the security policy of the platform is updated by including the security requirements of B.

Fig. 1. The S×C Workflow

The S×C checks will be run in case of bundle code update or bundle policy update. These checks, however, are variations of the installation scenario. Thus, in the sequel we will focus only on the installation scenario as the most representative one.

In terms of technical realization, the S×C framework can be easily integrated with the OSGi framework as a bundle, provided it is granted the permissions.

3 The Running Example

We consider as a case study an OSGi platform deployed as a service gateway in a smart home. A security-unaware user, typically a resident of the smart home, can download and install untrusted bundles onto the platform. The bundles are providing various services on the platform and can interact generating an added value for the user. If the framework can host multiple third-party bundles which can freely register services, the platform owner has to make sure that there are no security or functionality problems of the different bundles installed by the end user (who most likely does not even know what is a bundle and just sees the web interfaces of the services). Thus, a threat scenario under investigation is a

case when a bundle gains unauthorized access to the sensitive data of another bundle (*security threat*), or a bundle is malfunctioning due to unavailability of some external entity (*functionality threat*).

The running example was shared by an OSGi Alliance [1] member Telefónica[2]. Let us consider Alice, the smart home resident, and a telecom provider, the owner of the platform. Alice can download bundles for entertainment (news RSS feeds, media bundles from TV providers) or even bundles with traditional Internet content (like Facebook or Twitter), as new TV sets are used today for all these purposes. For the lack of space we use fictional names in the case study. The full version of the case study and more details about the S×C solution for the OSGi framework are available in the companion technical report [4].

Alice, a beginner stock market player, downloads and installs bundle A from provider $FSM.com$ that provides her with an interface of the stock market operations. This bundle includes service S_A that retrieves updates about the stock prices. However, Alice later finds and installs another stock market bundle B from $BH.fr$ provider, that provides a service for prices information retrieval S_B and a service S_{fr} that allows Alice to transfer money from her stock market account (registered on $BH.fr$) to her Happy Farm account on Facebook ($FB.com$). Thus, Alice also installs Happy Farm bundle F.

The bundle providers want to ensure that their security policies related to bundles and services usage are enforced on the Alice's platform. Their requirements are as follows:
FSM.com: *Access to S_A service is allowed only for bundles signed by $FSM.com$.*
BH.fr: *Access to S_B service is allowed only for bundles signed by $BH.fr$. Only bundles signed by $BH.fr$ can import the package containing S_B. Access to S_{fr} service can be granted only for bundles signed by $FB.com$ or by $BH.fr$.*

The OSGi platform at Alice's smart home has to ensure that the requirements of each provider are respected. Let us now discuss how the unmodified OSGi platform itself can enforce the requirements of the providers and why this approach is not satisfactory.

The OSGi Framework Background. Let us now present the relevant OSGi platform details [1]. An OSGi bundle is a JAR file that includes the manifest.mf file (*manifest file* in the sequel) containing the necessary OSGi metadata: the symbolic name of the bundle, its version, the dependencies and the provided resources. Some packages of a bundle can be *exported* (accessible for other bundles on the platform). A bundle also typically includes an *activator* (used for bootstrapping when the bundle is started) and a file with security permissions requested by the developer.

The OSGi system maintains the evolving lifecycles of the bundles. The bundle must first be *installed*. It can be *resolved* when all its *dependencies* are resolved. Bundles can depend on external entities (by requiring other bundles, an execution environment, a library, etc.). A bundle can start only after its dependencies

[1] http://www.osgi.org/
[2] http://www.telefonica.com/

were resolved. Bundles can express their dependencies as *requirements* on *capabilities*. Capabilities are attribute sets in a specific namespace and requirements are filter expressions that assert the attributes of the capabilities. A requirement is satisfied when there is at least one capability that matches the filter.

Bundles can interact through two complementary mechanisms: the export and import of packages and the service registration/lookup facility. A *service* is a normal Java object registered under a Java interface with the *service registry*. Bundles can register services, search for them, and receive notifications when the system state changes. A *service interface* is the specification of the service's public methods. A developer creates a service by implementing its service interface and registering it with the service registry. When requesting a service, a bundle specifies the name of the service interface and, optionally, a filter to narrow the search. In response, the framework first sends ServiceReference objects of the services that satisfy the search filter. The actual service object can then be acquired by passing the ServiceReference to the platform, provided the caller has the ServicePermission[ServiceReference, GET] permission.

Security of the OSGi platform is based on the Java 2 security architecture. Each bundle is associated with a set of permissions, that are queried at runtime. The OSGi specification defines ServicePermission, BundlePermission and PackagePermission, which are used for getting/registering a service, importing/exporting bundles and packages respectively. The platform can authenticate code by download location or by signer (digital signature). The Conditional Permission Admin service manages the permissions based on a comprehensive conditional model.

A bundle has a set of *local permissions* defined by the developer in the file permissions.perm (*permissions file* in the sequel). These are the actual detailed permissions needed by this bundle to operate. A framework also provides an administrative service to associate a set of *system permissions* with a bundle. The bundle's *effective permissions* are the intersection of the local permissions and the system permissions. That is, a bundle cannot get more permissions than its local permissions set. Thereby, a bundle developer can limit the possible permissions of a bundle, but she cannot require a minimum set of necessary permissions to be granted and she cannot directly influence the set of system permissions granted to the bundle.

Security Challenges. A confidentiality attack can be realized by the bundle A of provider $FSM.com$ getting access to the sensitive stock market prices service S_B of provider $BH.fr$. This might happen if A imports the package containing the service S_B definition, requires the bundle B (thus importing all its exported packages), or tries to get a reference to this service from the service registry and then get access to the object referenced.

The current OSGi security management suggests to address this security threat using the permissions system. Import of a package or a require-bundle action can be granted if the requiring bundle has corresponding permissions. Simple reviewing of the manifest file and permissions file of the bundle A can report about a (potential) attempt to interact with the bundle B. However,

there is no convenient and simple way for the owner of the bundle B, the $BH.fr$ provider, to declare which other bundles are allowed to import its packages.

Package importing can be guarded by the permissions mechanism. Currently only the platform owner (the telecom provider) can define and manage policies in the ConditionalPermissionAdmin service policy file. The $BH.fr$ provider might contact the telecom provider to ask him to set the required permissions, or its bundle B, being granted the necessary permissions, can add new permissions to the ConditionalPermissionAdmin policy file. These approaches are organizationally cumbersome and costly, as they require the operator to push the changes to its customers before any downloads of $BH.fr$ bundles, even if some customers have no intention of using them.

Another solution, that is traditional for mobile Java-based component systems, could be to ask Alice each time a specific permission is needed. But Alice is not the owner of the bundles to make such decisions, nor is she interested to do so.

The S×C paradigm, on the other hand, enables the bundle providers with a way to specify in the bundle contracts the necessary authorizations. The framework can then collect these authorizations from the bundles and incorporate them in the S×C policy on demand.

Service usage is more tricky though. Again, the necessary authorizations for the service usage (more precisely, GET permissions for service retrieval) can be delivered within bundle contracts and incorporated into the policy file of the system. But the invocations of the methods within a service, once the necessary reference is obtained, are not guarded by the permission check, and usually the security checks are placed directly within the service code, thus mixing the security logic with the execution logic.

Functionality Challenges. Let us now consider a more complex scenario.

Example 1 *Alice wants to install the Sims add-on from the EA.com provider. This add-on is packaged into the bundle C and it will provide an integration of the Happy Farm account with her the Sims account. The functional requirement of the EA.com provider is the following: "The bundle C can be installed if and only if the F bundle is available on the platform and provides the Happy Farm service S_F.*

The requirement in Example 1 means that bundle C can be installed only if the service S_F is already provided on the platform. This requirement prevents the denial of service by the Sims bundle, which can cause a restart of the whole system since the bundles are running on top of a single JVM. This functional requirement is, in fact, unsupported by the current OSGi specification. Requirements/capabilities model cannot provide guarantees on the provided services (except that their definition exists on the platform).

In the following sections we present our solution in detail, starting with the formal model of the current OSGi specification in the next section.

4 The OSGi Platform Formal Model

The entities on the OSGi platform are bundles and services, but the formal model also takes into account the lifecycle of bundles, as it is an explicit part of the OSGi platform.

Let $\Delta_{\mathcal{B}}$ be a domain of symbolic bundle names, $\Delta_{\mathcal{S}}$ be a domain of symbolic service names and $\Delta_{\mathcal{P}}$ be a symbolic domain of package names. Let also $\Delta_{\mathcal{L}}$ be a domain of location strings for the bundles and Δ_{Sign} be a domain of bundle signers names. We also define a set of local permissions requested by each bundle in its permissions file by $\mathsf{Permissions}_{\mathsf{bundle}}$, where each single permission perm is a pair $\langle Target, Action \rangle$.

Definition 1 (Bundle). *A bundle B is a tuple $\langle B, state(B), \mathsf{exports}_{\mathsf{B}}, \mathsf{imports}_{\mathsf{B}}, \mathsf{Permissions}_{\mathsf{B}}, \mathsf{Location}_{\mathsf{B}}, \mathsf{Signer}_{\mathsf{B}} \rangle$, where $B \in \Delta_{\mathcal{B}}$, $state(B) \in \{\mathsf{Installed}, \mathsf{Resolved}, \mathsf{Starting}, \mathsf{Active}, \mathsf{Stopping}\}$ is the current state in which the bundle resides at the moment, $\mathsf{exports}_{\mathsf{B}} \subseteq \Delta_{\mathcal{P}}$ and $\mathsf{imports}_{\mathsf{B}} \subseteq \Delta_{\mathcal{P}}$ are the sets of packages exported and imported by B correspondingly, $\mathsf{Permissions}_{\mathsf{B}}$ is the set of local permissions of bundle B, $\mathsf{Location}_{\mathsf{B}}$ is the download location and $\mathsf{Signer}_{\mathsf{B}}$ is the signer (the provider) of the bundle B.*

$\mathsf{Uninstalled}$ state is not considered, as the bundle is not functioning in this state and cannot be transferred to other states, so this state is equivalent to deletion of a bundle.

We define an *"is defined in"* relation $\diamond$ for packages, bundles and service interfaces. Let B be a bundle, S be a service interface and P be a package. We will denote by $P \diamond B$ the fact that P is a package defined in the bundle B and by $S \diamond P$ the fact that the service interface S is defined in the package P. For defining locations of bundles we define a $\vdash$ relation (*"comes from"*), we will denote as $L \vdash B$ the fact that bundle B comes from location $L \in \Delta_{\mathcal{L}}$. Also for locations we can define a notion of location inclusion, we will denote as $L_1 \subseteq L_2$ if the string location L_1 includes the string location L_2 as a prefix (without wildcards).

Definition 2 (OSGi Platform). *The platform Θ is a tuple $\langle \mathcal{B}, \mathcal{S}, \mathcal{R} \rangle$ where $\mathcal{B}$ is a set of bundles on the platform, $\mathcal{S} \subseteq \Delta_{\mathcal{S}}$ is a set of services on the platform, and $\mathcal{R} \subseteq \mathcal{B} \times \mathcal{S}$ is a service provision relation such that for each service $S \in \mathcal{S}$ there exists only one bundle $B \in \mathcal{B}$ such that the pair $(B, S) \in \mathcal{R}$, $state(B) \in \{\mathsf{Active}, \mathsf{Starting}, \mathsf{Stopping}\}$ and $S \diamond P$ such that $P \diamond B$ and $P \in \mathsf{exports}_{\mathsf{B}}$.*

Thus, in the model we consider that a service can be provided only by a bundle in an appropriate state that exports a package containing the definition of this service. We will denote the fact that bundle B can provide service S (exists package P such that $S \diamond P$ and $P \in \mathsf{exports}_{\mathsf{B}}$) as $S@B$.

We want to ensure that bundles interact on the platform in compliance with the pre-defined security policies set by the bundle owners. Thus, we start with the definition of bundle interaction. Informally, two bundles interact if one of them imports an exported package from another, or consumes a service provided by the other bundle.

Definition 3 (Bundle Interaction). *Let $B_1, B_2 \in \mathcal{B}$. We say that B_1 interacts with B_2, denoted $B_1 \bowtie B_2$ if at least one of the following conditions is satisfied:*

- *Exists $P \in \mathsf{exports}_{B_1} \cap \mathsf{imports}_{B_2}$ such that $P \diamond B_1$ – there exists a package exported by B_1 and imported by B_2;*
- *Exists a local permission* $\mathsf{perm} = \langle \mathsf{GET}, S \rangle \in \mathsf{Permissions}_{B_2}$ *where S is a service such that exists a package P: $S \diamond P$ and $P \in \mathsf{exports}_{B_1}$ – there exists a local permission of bundle B_2 to get a service reference from bundle B_1.*

This definition captures *a potential direct control flow among bundles.*

Functionality guarantees on the OSGi platform can vary. An interesting scenario was discussed in Example 1, where a bundle wants to have some functional requirements fulfilled prior to be loaded. The capabilities approach currently explored on the OSGi platforms is purely static and declarative. We want to enhance it with the dynamics of evolving platforms and with the guarantees given by the framework itself rather than by (potentially untrusted) bundles.

In Example 1, the C bundle wants to be installed on the platform only if a specific service is already provided there. While the presence of the service interface definition can be (limitedly) ensured by the capabilities approach, only the platform itself can assure that the service is indeed provided, or a certain bundle is in a desired state, or that a competitor's bundle is not installed at all. If later, when the bundle will already be installed, the platform will evolve such that the desired service will be unregistered or an undesired bundle appear, the bundle can be notified about it through the event system and take the actions it needs to protect itself (be removed, stop, notify the provider, etc.).

There is also another interesting problem that can be considered. The primary purpose of the requirement-capabilities model is to provide an explicit assertion about the environment before a bundle becomes active and its code starts to run. This prevents bundles that cannot run because they are not suitable for a given environment from becoming active, or even installed, when this header is used by a management system. The S×C framework can become the management system that will assure bundles they will never enter even the Installed state on a platform that is not suitable for them.

Contracts and Policy. The claim of a bundle (sufficient to cover the security and functionality issues discussed above) can be easily extracted from the bundle's manifest file and permissions file. Thus, the ClaimExtractor component duties will be to extract this information. The policy of a bundle is a new component specified in the contract that requires a permission notation and a notation for functional requirements.

Definition 4. *Let B be a bundle. The* $\mathsf{Contract}_B$ *is a tuple* $\langle \mathsf{sec.rules}_B, \mathsf{func.rules}_B \rangle$*, such that:*

- $\mathsf{sec.rules}_B$ *is a set of permissions of the form $\langle Action, Target, Authorized\ entity \rangle$, that specifies the security policy on the usage of B's packages and services, where*
  - *Action* $\in \{\mathsf{IMPORT}, \mathsf{GET}\}$*;*

– $Target \in \bigcup_{P \diamond B} P \cup \bigcup_{S@B} S$;
– $Authorized\ entity \in \Delta_{\mathcal{B}} \cup \Delta_{\mathcal{L}} \cup \Delta_{Sign}$;

• $\mathsf{func.rules_B}$ *is a set of functional requirements of* B *of the form* $\langle Desired\ state, Flag, State \rangle$*, that specifies the requests of the bundle for functionality available on the platform, where*
– $Desired\ state \in \{Present, Not\ present\}$;
– $Flag \in \{Bundle, Package, Service\}$;
– *State differs in the following fashion:*
· $Flag = Bundle$, *then* $State \in \{\mathsf{Installed}, \mathsf{Resolved}, \mathsf{Starting}, \mathsf{Active}, \mathsf{Stopping}\}$;
· $Flag = Package$, *then* $State \in \{Present, Exported\}$;
· $Flag = Service$, *then* $State \in \{Present, Provided\}$;

Using the above notations bundles can express various security and functional requirements on other bundles on the platform. Bundles can be installed on the platform if and only if all their security and functional requirements are satisfied and their behavior is compliant with the policies of all other bundles on the platform.

We propose the bundle contract to be delivered within its manifest file by using the possibility to define new manifest file headers in the common header syntax. The newly defined headers processed by the S×C manifest file parser (the ClaimExtractor), are **sxc-secrules** and **sxc-funcrules**. The first header specifies the bundle's sec.rules separated by commas (elements of permissions are separated by colons). The second header denotes the bundle's func.rules separated by commas (elements of requirements are separated by colons). The security policy of the platform is defined as follows.

Definition 5 (Security Policy of the Platform). *For a platform* Θ *its security policy* $\mathsf{Policy_\Theta} = \bigcup_{B \in \mathcal{B}} \mathsf{sec.rules_B}$.

Example 2 *Let us consider the running example from Section 3. The bundles A, B and F are installed on the platform. The security policy* $\mathsf{Policy_\Theta}$ *of the Alice's platform equals to* $\{\mathsf{sec.rules_A}, \mathsf{sec.rules_B}, \mathsf{sec.rules_F}\}$. *The contracts of the bundles can be derived from their requirements Thus,* $\mathsf{sec.rules_F} = \{\emptyset\}$. $\mathsf{sec.rules_A} = \{\langle \mathsf{GET}, S_A, FSM.com \rangle\}$. $\mathsf{sec.rules_B} = \{\langle \mathsf{GET}\ , S_B, BH.fr \rangle, \langle \mathsf{IMPORT}, P_B, BH.fr \rangle, \langle \mathsf{GET}, S_{fr}, BH.fr \rangle, \langle \mathsf{GET}, S_{fr}, FB.com \rangle\}$.

For a platform Θ its *functional state* is at any given moment of time defined by the platform itself: the installed bundles and provided services.

The* S×C *Checks.

Definition 6. *Let* Θ *be an OSGi platform and* B *is a loaded bundle. We say* Θ *can host* B *securely iff the following conditions are satisfied:*

• ***Stable Security****. For all bundles* $A \in \mathcal{B}$ *if* $A \bowtie B$ *then a corresponding permission for* B *exists in* $\mathsf{sec.rules_A}$*, and if* $B \bowtie A$ *then a corresponding permission for* A *exists in* $\mathsf{sec.rules_B}$.

• ***Stable Functionality****. All functional requirements described in* $\mathsf{func.rules_B}$ *are satisfied by* Θ.

We note that the stable functionality property may not hold for some other bundle A immediately after the installation of bundle B on the platform. However, A will be notified about the situation and can take appropriate actions.

These are some of the checks (full list can be found in [4]) to be executed by the PolicyChecker. If B imports a package P of A, the check is "$\langle \mathsf{IMPORT}, P, B\rangle \in \mathsf{sec.rules_A}$" (the source of information is the manifest and the permissions files of B). If B requires a bundle A to be present on the platform in Active state ($\langle Present, Package, P, Exported\rangle \in \mathsf{func.rules_B}$), the check is "in the current functional state exists bundle $A \in \mathcal{B}$ such that $state(A)$=Active". It can be easily demonstrated (proof by cases) that if the necessary checks for a new bundle B are performed by the S×C framework, then the platform can host B securely.

We can note here that the permissions file could be a weak source of information, as it may only require AllPermission, letting the system to define the upper bound of permissions for the bundle. This problem can be solved by awareness of the developers that their bundles will be rejected if the required permissions will be too demanding.

5 Evaluation

The evaluation of the proposed framework was conducted in collaboration with industry experts from Telefónica. Two top-level criteria were defined: effective usage (which includes applicability and human effort), and specific industrial criteria (level of automation).

Applicability. The OSGi framework presents no obstacles for the implementation of such a system as a bundle.

Human effort. It is expected that the basics of S×C methodology will require a relatively low effort for a developer to learn how to create contracts. We estimate that the deployment of S×C will require very little effort for the operator of a network of home gateways. Installing a bundle (such as the S×C framework) on an OSGi platform is typically a routine task that presents few challenges. The S×C deployment should be transparent to and require no effort from the users of a gateway, except from perhaps providing confirmation for a system update.

Automation. The S×C is expected to work on a fully automated manner, inspecting bundle contracts and enforcing contract policies without need for user interaction. The generation of contracts, however, cannot be automated in the current framework. As a future development, it would be interesting to find ways to automate contract generation, or at least to have automated tools that guide developers in the process.

Overall, the current S×C specification for OSGi platforms looks like a promising starting point, which has some immediate applications as well as some very interesting research lines which will need to be studied in depth. One significant upside of the S×C methodology is the fact that it's optional and backwards compatible, which means that service providers do not need to immediately incorporate contracts into each running service as soon as S×C is adopted, but

can introduce them gradually as new bundle versions are released. This allows careful planning of S×C contracts and better distribution of the workload.

6 Related Work

Though the OSGi technology is gaining popularity today, there are not so many papers dedicated to the OSGi security. Moreover, these papers are not applicable to the scenarios and threats we considered[3].

The proposal by Parrend and Frénot [10] on installation time verification for the OSGi bundles is the closest to our current work. The authors advocate the installation time static analysis of bundle code that could replace the time-consuming run-time checks for given permissions. Their Component-based Access Control (CBAC) mechanism allows to parse the bytecode of the loaded bundle and check that all the sensitive methods it invokes are allowed. The testing of the CBAC tool concluded that the overhead of the installation time verification is insignificant in comparison with the run-time checks. However, the drawbacks of the approach are the same as were concluded for the OSGi framework: the bundle providers are still not entitled to defining their own policies.

There are a number of permission-based security frameworks for service platforms outside of the OSGi community. We can mention the works of Enck et al. [6] and Nauman and Khan [7] on the loading time mobile code certification for Android. While strengthening the security of the framework, these approaches, in contrast to our solution, leave the non-trivial task of defining the policy for the user and can result in non-functioning software. The needs of the Android applications to be able to regulate how they interact with other applications were advocated by Ongtang et al. [9]. They have proposed Saint – an extension of the Android platform that governs installation time permission assignment and controls the run-time use of permissions.

The Security-by-Contract paradigm that had inspired our proposal was investigated and implemented by Bielova et al. for mobile Java-based devices [3] and by Dragoni et al. for the Java Card platforms [5]. Our current work improves previous approaches by adapting the S×C scheme to the systems with permission-based security management, thus allowing it to be easily adopted on similar platforms (e.g., Android, Chrome).

7 Conclusions

In this paper we have presented a Security-by-Contract paradigm for the OSGi platforms. We gave an overview of the OSGi framework and described how the the OSGi platforms can benefit from the S×C approach. The paper contains the following contributions: (1) the proposal how to enable the bundle providers with ability to effectively express their security and functional requirements on the

[3] While here we present only the most related work, a more thorough review can be found in the full version of the paper [4].

platform; (2) a formal model of an OSGi platform and the notations for security and functional requirements of bundles; (3) the S×C architecture for load time verification on OSGi.

The main benefits that the S×C approach can bring to OSGi platforms are the following. From the security aspect, the bundle providers can now specify the authorizations for access to their bundles, packages and services. The policies can be updated easily and the update does not require an interaction from the platform owner, an access to the framework policy file or an update of the execution logic of the bundle. For the functionality aspect, the bundle providers have now a more powerful tool for expressing their functional requirements than the requirement/capability model of OSGi. The contracts can express requirements on the current state of the platform (including requirements on the states of the bundles or certain services provision, or absence of the competitor's resources).

Acknowledgements. We would like to thank Telefónica for sharing the case study and evaluating our proposal. This work was partially supported by the EU under grants EU-FP7-FET-IP-SecureChange (grant n. 231101), FP7-IST-NoE-NESSOS (grant n. 256980) and EU-FP7-ICT-IP-ANIKETOS (grant n. 257930).

References

1. The OSGi Alliance. OSGi service platform core specification. Version 4.3, 2011.
2. P. Belimpasakis, M. Michael, and S. Moloney. The home as a content provider for mash-ups with external services. In *CCNC'2009*, pages 1 –5, 2009.
3. N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Matching in security-by-contract for mobile code. *Journal of Logic and Algebraic Programming*, 78(5):340 – 358, 2009.
4. P. Capelastegui, O. Gadyatskaya, F. Massacci, and A. Philippov. Security-by-Contract for the OSGi framework. Technical Report DISI-12-002, DISI, University of Trento, Italy, `http://www.disi.unitn.it/~gadyatskaya/osgi.html`.
5. N. Dragoni, E. Lostal, O. Gadyatskaya, F. Massacci, and F. Paci. A load time Policy Checker for open multi-application smart cards. In *Proc. of POLICY'11*, pages 153–156.
6. W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *CCS 2009*, pages 235–245. ACM, 2009.
7. M. Nauman and S. Khan. Design and Implementation of a Fine-grained Resource Usage Model for the Android Platform. In *IJAIT*, 2010.
8. A. Ngu, M. Carlson, Q. Sheng, and H. Paik. Semantic-based mashup of composite applications. *IEEE Tran. on Services Computing*, 99:2–15, 2010.
9. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. In *Proceedings of ACSAC 2009*, pages 340–349, 2009.
10. P. Parrend and S. Frénot. Component-based access control: Secure software composition through static analysis. In *Software Composition*, LNCS 4954, pages 68–83. Springer, 2008.
11. P. Phung and D. Sands. Security policy enforcement in the OSGi framework using aspect-oriented programming. In *COMPSAC'2008*, pages 1076–1082, 2008.