Log-based Lazy Monitoring of OSGi Bundles

Giulio Caravagna\textsuperscript{1}, Gabriele Costa\textsuperscript{2**}, Giovanni Pardini\textsuperscript{3}, and Luca Wiegand\textsuperscript{2}

\textsuperscript{1} Dipartimento di Informatica, Sistemistica e Comunicazione, Università degli Studi di Milano-Bicocca, Italy
\texttt{giulio.caravagna@disco.unimib.it}

\textsuperscript{2} Institute for Informatics and Telematics, National Research Council, Pisa, Italy
\texttt{\{gabriele.costa,luca.wiegand\}@iit.cnr.it}

\textsuperscript{3} Dipartimento di Informatica, Università degli Studi di Verona, Italy
\texttt{giovanni.pardini@univr.it}

Abstract. Lazy controllers are execution monitors which do not continuously observe the behaviour of their target. Monitors are activated and deactivated according to a scheduling strategy. When a lazy controller is activated, it checks the current security state and, in case of a violation, terminates the execution. Otherwise, if the current execution trace is safe, the monitor is suspended and its activation is scheduled again. The inactivity period is computed by considering the risk that, from the current state, the target can produce a security violation.

In this paper we present a prototype using existing logging API, i.e., the Commons Logging Package, for remotely watching the execution of OSGi bundles. We claim that our solution can efficiently follow the target system keeping under control the delay in detecting violations. Also, as we use standard OSGi platform and facilities, we show that our monitors can run under very realistic assumptions for bundle applications.

1 Introduction

Security monitors are commonly used for controlling that a target respects some security requirements. Many authors proposed important contributions to the theory and practice of security controllers, e.g., see [12,3,5,16,9]. All these approaches present security controllers that can guard program executions and run reaction procedures. Several other proposals, e.g., see [13,2,6,10], also exploit a verification step for supporting the synthesis and execution of security monitors.

Many of the most influential proposals for modelling security controllers, e.g., see [12] and [8], rely on a continuous, step by step observation of the target execution. Although this is a reasonable approach when monitoring the local

\textsuperscript{*} Work partially supported by EU-funded projects FP7-231167 Connect, FP7-257930 Aniketos, FP7-256980 NESSoS and by FP7-257876 SPaCioS.

\textsuperscript{**} Now at Dipartimento di Informatica, Sistemistica e Telematica, Università di Genova, Italy.
execution of a program, it may be difficult or even impossible to implement the same strategy when the target is expected to execute remotely, e.g., consider the Software as a Service (SaaS) paradigm. Also, application monitoring usually requires some modification to either the target program, e.g., code instrumentation, or the execution platform, e.g., system calls wrapping. These techniques do not fit with the remote execution scenario where the mobile code is digitally signed and the execution platform must comply with standard specifications.

For these reasons, we presented a new class of security controllers, namely lazy controllers [4]. Like standard controllers, lazy controllers watch their target execution. However, unlike standard monitors, they can autonomously decide to suspend the observations for a certain time span. Clearly, in this way, a lazy controller could miss the observation of a security violation while it is suspended.

Hence, a crucial aspect of the applicability of lazy controllers is the definition and the calculation of the “risk” deriving from pausing the controller guarding the target. A good scheduling for the observations can prevent unobserved security violations, but there is no general, non-trivial way of finding such a scheduling. Intuitively, minimizing the possibility of having a bad scheduling is the main issue when using lazy controllers.

Being able to asynchronously control the target activity has some advantages. In terms of performance and costs, for instance, the monitoring process can be optimised by reducing the number of validity checks on the target behaviour. Another important advantage is in terms of applicability. Indeed, our controllers can be implemented by using existing facilities, while most other approaches use ad-hoc solutions as discussed above. For instance, log auditing [7,1] is often used to asynchronously check the last actions performed by a system.

In this paper we present an implementation of our lazy controllers for the execution monitoring of Java OSGi bundles [15]. Intuitively, we remotely monitor the execution of a bundle by inspecting its execution log. We assume bundles to use the Common Logging API [14] for writing their execution log. Then, we execute the lazy monitor on a different platform. The lazy monitor can request to the bundle execution platform an instance of its log, i.e., a plain sequence of security operations performed by the bundle. When a violation is discovered, the monitor changes the status of its target from active to stopped.

We show that our method offers substantial advantages w.r.t. a standard security monitor applying the same security policy. These advantages are mainly in terms of performances, i.e., we produce a significantly lower overhead on the system, and applicability, i.e., we can use our approach under very realistic assumptions. As a matter of fact, every OSGi platform provides some logging facilities to installed bundles.

The paper is structured as follows. In Section 2 we briefly introduce lazy controllers and their features. In Section 3 we discuss our prototype implementation and its behaviour and Section 4 concludes the paper.
2 Lazy Security Controllers

We briefly recall the theory of lazy security controllers [4]. A security monitor is a tuple \((\Sigma, C, \implies)\) where \(C\) is a set of states and \(\implies\) is a transition relation triggered by \(\Sigma\) actions. We write \(C \implies S \implies' C'\) to denote that the composition of a system in state \(S \in S\), where \(S\) is the set of states of the target, with a controller in state \(C\) performs a visible action \(\alpha\). The new system and controller states are \(S'\) and \(C'\), respectively. Given a discrete/continuous time domain \(T\) a lazy controller is defined as follows.

**Definition 1 (Lazy Controller).** A lazy controller is \((\Sigma, C, \implies, \implies_{\uparrow}, \zeta)\) where:

- \(\implies_{\uparrow} \subseteq (T \times C \times S) \times (\Sigma \cup \{\cdot\}) \times (T \times C \times S)\) is the active monitoring relation;
- \(\implies_{\uparrow} \subseteq C \times \Sigma \times C\) is the update relation for unseen actions;
- \(\zeta : C \times T \to T\) is the scheduling function.

Where \(\Sigma = \{\bar{a} \mid a \in \Sigma\}\) is the set of unseen actions.

Relation \(\implies_{\uparrow}\) is the operational notion of activity logging: while the controller is not observing the system, i.e., it is idle, every action \(a \in \Sigma\) performed by the target is logged as unseen, i.e., it is \(\bar{a}\) and is freely performed by the target. Instead, function \(\zeta\) is used to schedule the observations over the execution of the target. We assume that the controller states and time allow for the evaluation of such a sensitive information. In [4] the synthesis of lazy controllers for non-deterministic timed systems with non-instantaneous actions and for both discrete-time and continuous-time markovian probabilistic systems is discussed. In each case the (analytical) probability that a lazy controller misses the detection of a violation is given.

Let \(D\) be the set of all the configurations of the form \(T \times C \times T \times S \times T\) and let \(A = \Sigma \cup \Sigma \cup \{\cdot\}\). A Labelled Transition System (LTS) is a graph with
states and labelled edges between states. States denote system configurations and edges transitions from a configuration to another. The semantics of a lazy controller is a LTS \((D, A, \rightarrow_{\text{lazy}})\) where \(D\) is the set of states, \(A\) is the set of labels and \(\rightarrow_{\text{lazy}} \subseteq D \times A \times D\) is the least transition relation defined by the inference rules of Figure 1. The rules are given in the form premises \(\rightarrow\) conclusion, along the lines of the Structural Operational Semantics approach [11]; here we informally comment on the rules (see Fig. 1) and we refer the interested reader to [4].

- **(Sleep)** states that, if at time \(t\) the controller is active, denoted by \([C]_0\), and the next observation is scheduled at time \(t+k\), then the controller can idle till that time, here denoted as \([C]_k\). The transition label \(\cdot\) means that this derivation does not involve any action of the target.
- **(Mon)** applies when the controller is actively following the target. As the scheduler prevents the monitoring from becoming idle, i.e., \(\zeta(C, h) = 0\), any action of the target started at \(t-h\) and completing at \(t-h+x\) is monitored. Relation \(\Rightarrow\) characterizes this behaviour of the controller.
- **(Log)** states that, if the time is \(t\) and the controller has scheduled the next observation at time \(t+k\), then any action which the target \(S\) performs before \(t+k\) is not controlled, but simply logged by means of the derivations using \(\rightarrow_{\text{up}}\) transition. In this time-window a violation may happen, not being detected up to time \(t+k\);
- **(Wake)** allows the controller to spend time autonomously and synchronously with the target \(S\).

Lazy controllers include standard security controllers at the semantic level, as stated in the following theorem whose proof can be found in [4].

**Theorem 1 ([4]).** Let \((\Sigma, C, \Rightarrow)\) be a security controller, let \((\Sigma, C, \Rightarrow, \rightarrow_{\text{up}}, \zeta)\) be the lazy security controller with \(\rightarrow_{\text{up}}\) arbitrarily defined and \(\zeta : C \times T \rightarrow \{0\}\). Then, for any target \(S \in S\) and time \(t \in T\)

\[
\langle t, C \triangleright S \rangle \xrightarrow{\Rightarrow} \langle t', C' \triangleright S' \rangle \quad \text{iff} \quad \langle t, [C]_0 \triangleright [S]_0 \rangle \xrightarrow{\rightarrow_{\text{lazy}}} \langle t', [C']_0 \triangleright [S']_0 \rangle.
\]

In words, Theorem 1 says that, forcing a lazy controller to be always active, i.e. \(\zeta(C, t) = 0\) for any \(C\) and \(t\), we obtain the same enforcement process produced by the corresponding security controller.

### 3 Prototype implementation and discussion

In this section we present our prototype and we discuss its behaviour and performance. In order to run our prototype under realistic settings, we defined a case study in which a service running on a remote OSGi platform is monitored.

**Case study.** We consider a simple medical prescription service infrastructure. The system consists of four actors: (i) a prescription service, (ii) doctors, (iii) pharmacies and (iv) a delivery service. Registered doctors can use the prescription service to fill prescription forms for their patients and submit them to a pharmacy or to the delivery service. Briefly, the program works as follows:
Fig. 2: The prescription system FSM.

Fig. 3: The privacy policy FSM.

1. initially, the system waits for users, i.e., doctors, to log in (login);
2. then the doctor can add one or more medicines (standard, i.e. add_med, or HIV-specific, i.e., add_hiv) to the prescription;
3. finally, the doctor chooses between two modalities, either pharmacy or deliver, which specify how the patient gets access to the medicines.

At each step, the doctor can cancel (cancel) the operation and, at the end, he must confirm (confirm) the prescription. Figure 2 shows the finite state machine (FSM) representing the prescription system.

In order to avoid privacy violations, HIV therapies must always be delivered at the patient’s residence. The FSM of Figure 3 represents the privacy policy described above. Briefly, the policy reaches the final state, i.e., detects a violation, if a session in which add_hiv has been invoked concludes with pharmacy.

Prototype structure. The OSGi bundle implementing the prescription service mainly consists of a simple RMI interface. The interface declares a method for each action labelling the FSM of Figure 2, e.g., deliver() for deliver. Each method behaves according to its specification, e.g., add_med() adds a medicine to the current prescription, and writes a new entry in the log. Logging functionalities are provided by an implementation of the org.apache.commons.logging.Log interface that simply appends the given label and a timestamp to a text file.

The lazy controller is an external application, i.e., running on a different platform w.r.t. the target service. At each control cycle, the monitor wakes up and requests the (fragment since the last request of the) current log to the remote platform. Then, the log trace is processed by the policy automaton in Fig. 3 to check if a violation has occurred. In case of a violation, the monitor sends a security error signal to the execution platform (here causing the target to be reinitialised). On the other hand, if the observed trace is legal, the lazy monitor just schedules the next control cycle and hibernates.

The scheduling function maps a pair of states $p$, for the target, and $q$, for the policy, into a hibernation time $t_{p,q} \in \mathbb{R}^+$. We compute hibernation times before starting the monitoring process. In this way, we carry out the computation only once and we store the pairs $\langle (p, q), t_{p,q} \rangle$ in a two-column table. Hibernation
times are computed using the procedure detailed in [4], starting from a description of the target system. Clearly, the behaviour of the system depends on the users/doctors. We assume that standard behaviour is known, e.g., by analysing the system execution. In our model we used two different descriptions: Continuous Time (CTMC) and Discrete Time Markov Chains (DTMC). In particular, we use the following matrices for describing the standard execution of the service:

\[
R = \begin{bmatrix}
0 & 1/30 & 0 \\
1 & 2/5 & 1/4 \\
2 & 0 & 0
\end{bmatrix}
\]

\[
P = \begin{bmatrix}
0 & 1 & 0 \\
1/20 & 17/20 & 1/10 \\
0 & 0 & 1
\end{bmatrix}
\]

The matrices describe the expected behaviour of the FSM of Figure 2. Matrix \(R\) contains rates of state transitions, corresponding to the parameters of exponentially distributed random variables, while \(P\) contains the probabilities of state transitions. Intuitively, time rates define the expected number of state transitions per second, e.g., \(R[1, 2] = 1/30\) means that a transition from state 0 to state 1 happens, on the average, every 30 seconds. Instead, the elements of \(P\) describe the probability of moving from the current state to the next one, e.g., \(P[2, 3] = 1/10\) means that state 3 has 1/10 probability to be the successor of 2. Also, note that \(R\) and \(P\) can collapse the values for more than one transition in a single value, e.g., \(P[2, 2] = 17/20\) denotes both \(\text{add\_med} (P_{\text{add\_med}} = 4/5)\) and \(\text{add\_hiv} (P_{\text{add\_hiv}} = 1/20)\) transitions.

**Performance evaluation.** The prescription service was developed with Eclipse Helios SR2 and executed on OSGi platform Equinox 3.3. Log libraries have been developed implementing the Apache Commons Logging API version 1.1.1.

We tested our system by automatically generating customer sessions of several types. Customers access the system which is monitored using a lazy controller. We synthesize the lazy controllers using the two matrices \(R\) and \(P\) introduced above and considering four different risk factors, i.e., 0.01, 0.05, 0.1 and 0.2. Also, we compared our monitors with a lazy controller which uses a scheduling function that returns the duration of the shortest path leading to a violation from the current state, computed by means of the Dijkstra algorithm (note that the relation between this scheduling function and the notion of “risk” is not defined in general and, also, the scheduling times cannot be refined). For this purpose, we considered the matrix \(R'\) such that \(R'[i, j] = R[i, j]^{-1}\) (and \(R'[i, j] = \infty\) if \(R[i, j] = 0\)).

For the overhead analysis we considered customers that statistically behave in a compliant way with respect to the original specification, i.e., the behavioural matrices. The execution overhead is a measure of the computational effort due to the monitoring activity in comparison with the computation of the target. For the continuous time model we considered the activity time of the monitor against the overall execution interval. Instead, for the discrete time model we compared the number of controller synchronizations and the total number of service invocations. Figure 4a shows the simulation output.

As expected, both the approaches increase their performance with the growth of the risk threshold. Moreover, in general they perform better than the Dijk-
the applicability of security monitors to many real-world scenarios. Moreover, technique schedules security checks during its execution rather than controlling

4 Conclusion

Fig. 4: Monitoring performance evaluation.

stra algorithm-based solution (dashed line). Clearly, such version does not gain advantage from the risk modification.

In order to test delays in violations detection, we executed our system with clients that only emit illegal traces (in the sense of Figure 3). The violating traces are generated using the same probabilities and rates of standard clients. Figures 4(a) and 4(b) show the violation detection delays observed in our tests.

Note that the delays for CTMC and DTMC-based monitors have completely different meanings and must be interpreted. Indeed, CTMC controllers work under real time settings, i.e., the monitor is created for keeping under control the time delay of a violation detection rather than the number of actions. Conversely, DTMC controllers aim at minimising the number of actions after a violation. However, it is interesting to compare the two models in both cases.

Finally, we also introduced an error factor for testing the stability of our solution. In particular, we considered users that do not perfectly comply with the given specifications, i.e. the matrices $R$ and $P$. Interestingly, we found that the performance and delay of our system are stable even with errors up to 30%.

4 Conclusion

We presented a prototype implementation of a monitoring environment using lazy controllers for verifying the policy compliance of a remote execution. Our technique schedules security checks during its execution rather than controlling the target continuously. Although this generates a risk factor, it also extends the applicability of security monitors to many real-world scenarios. Moreover,
we have shown that the risk of a security violation can be analysed and kept under control through the execution parameters of the controllers.

Lazy controllers are generated starting from the specification of a standard security controller. Then we add time constraints to the application rules. In this way, we can convert any existing security controller to a corresponding lazy one. This amounts to say that we can apply our solution to existing enforcement environments without redesigning them.

Finally, we considered the performances of our prototype under several settings and we showed execution statistics. The prototype was executed with realistic setting and applied to a case study using OSGi technology. All the experiments highlighted the flexibility and efficiency of our solution.

References