Lazy Monitoring for Distributed Computing Environments

Giulio Caravagna  
*Dipartimento di Informatica, Sistemistica e Comunicazione*  
Università degli Studi di Milano-Bicocca  
giulio.caravagna@disco.unimib.it

Gabriele Costa* and Luca Wiegand  
*Istituto di Informatica e Telematica*  
Consiglio Nazionale delle Ricerche  
name.surname@iit.cnr.it

Giovanni Pardini  
*Dipartimento di Informatica*  
Università degli Studi di Verona  
giovanni.pardini@univr.it

Abstract—Lazy controllers are a class of execution monitors that 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. Instead, 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 reach a faulty configuration. This behaviour is particularly interesting for systems which is difficult to monitor with standard solutions, e.g., web services.

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 in the context of web services.

Keywords—Security monitoring; distributed computing; mobile code;

I. INTRODUCTION

Security monitors are commonly used for controlling that a target, e.g., a program, respects a security requirements. Many authors proposed important contributions to the theory and practice of security controllers, e.g., see [18], [6], [8], [23], [13]. All these approaches present security controllers that can guard program executions and run reaction procedures. Also, several recent proposals, e.g., see [19], [5], [10], [14], exploit a verification step for supporting the synthesis and execution of security monitors.

Two of the most influential models for security monitors are [18] and [12]. The first work, originally proposed by Schneider, presents a category of Finite State Automata (FSA), namely security automata, for specifying security policies. Having a formal operational semantics, security automata can be directly used, as a template, to implement security monitors. Also, security automata are known to be expressive enough to represent safety properties [1], which are a crucial class of security properties.

The second proposal defines security controllers, called edit automata, which can enforce a larger class of security policies, namely edit policies. Edit policies are defined through the controllers which are in charge of enforcing them. Roughly, an edit automaton reads the next action of its target and decides whether to (i) allow it, (ii) suppress it or (iii) anticipate it with another one. Note that the enforcement can be used to simulate the behaviour of a security monitor, that is, edit policies are a proper superset of Schneider’s policies. Indeed, we can implement an enforcement strategy terminating the target by, for instance, appending to the trace a special action when a violation arises.

Although these are reasonable approaches when monitoring the local execution of a program, it may be difficult or even impossible to implement the same strategy when the target is expected to execute remotely. Hence, these methods are inherently weak when applied to distributed computing contexts, e.g., in Service Oriented Architecture (SOA). 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 present a new class of security controllers, namely lazy controllers [7]. 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 [11], [4] is often used to check the last actions performed by a system without interrupting its execution.

In this paper we present an implementation of our lazy controllers for the execution monitoring of Java OSGi bundles [22]. Intuitively, we remotely monitor the execution of a bundle by inspecting its execution log. We assume bundles to use the Common Logging API [21] 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 forces the termination of its target by changing its state 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 II we recall lazy controllers and their features. In Section III we discuss our prototype implementation and its behaviour and Section IV concludes the paper.

II. LAZY SECURITY CONTROLLERS

Here we briefly recall the theory of lazy security controllers and we provide some basic definitions. Lazy security controllers are built starting from an existing security monitor $C$. A security monitor has a set of states $C$ and a transition relation $\Rightarrow$ defining the way it reacts to the observed actions. We write $C S \Rightarrow C' > S'$ to denote that the composition of a system in state $S$ with a controller in state $C$ performs a visible action $\alpha$, and that the new system and controller states are $S'$ and $C'$, respectively.

Given a generic time domain $T$, which can be either discrete or continuous, and given a set of actions $\Sigma$, we formally define lazy controller as follows.

**Definition 1 (Lazy Controller).** A lazy controller is a tuple $(\Sigma, C, \Rightarrow, \rightarrow_{up}, \zeta)$ where:

- $\Rightarrow \subseteq (T \times C \times \Sigma) \times (\Sigma \cup \{\cdot\}) \times (T \times C \times \Sigma)$ is the active monitoring relation;
- $\rightarrow_{up} \subseteq C \times \Sigma \times C$ is the update relation for unseen actions;
- $\zeta : C \times T \rightarrow T$ is the scheduling function.

Where $\Sigma = \{ \hat{a} \mid a \in \Sigma \}$ is the set of unseen actions and $T$ is the time domain.

Relation $\rightarrow_{up}$ captures the operational notion of activity logging: as far as the controller is not observing the system, i.e. it is idle, every action is freely performed by the target and is logged. In this sense, an action $a \in \Sigma$ is unseen, i.e. it is $\hat{a}$, once it is freely performed by the target but is not observed by the controller.

Instead, function $\zeta$ provides the scheduling of the observations over the execution of the target. Notice that $\zeta$ is a function of the states of the controller and the time of the last action performed by the target, subsuming that the states of the controller, together with a information on time, contain enough knowledge to evaluate such a sensitive information.

Finally, in [7] 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 an analytical characterization of the probability that the lazy controller misses the detection of a violation is given. In the next sections we deal with the synthesis of lazy controller and we give intuitions on both structuring a controller and defining the sleep function.

Let $D$ be the set of all the configurations of the form $T \times C \times T \times S \times T$ and let $\Lambda = \Sigma \cup \Sigma \cup \{\cdot\}$. A Labelled Transition System (LTS) is a graph with states and labelled edges between states. Each state denotes a configuration of the system under consideration and each edge denotes a transition from one configuration, called source, to another, called destination. The semantics of a lazy controller is a LTS $(D, A, \rightarrow_{lazy})$ where $D$ is the set of states, $A$ is the set of labels and $\rightarrow_{lazy} \subseteq D \times A \times D$ is the least transition relation defined by the inference rules of Figure 1. The rules are given in the form $\text{premise} \rightarrow \text{conclusion}$ along the lines of the Structural

$\begin{align*}
\text{(Sleep)} & \quad \zeta(C, h) = k \quad k > 0 \\
& \quad \langle t, [C]_0 \bowtie [S]_h \rangle \rightarrow_{lazy} \langle t, [C]_k \bowtie [S]_h \rangle \\
\text{(Mon)} & \quad \langle t', C \bowtie S \rangle \xrightarrow{\alpha_{\text{lay}}} \langle t' + x, C' \bowtie S' \rangle \\
& \quad h \leq x \\
& \quad \langle t - h, S \rangle \xrightarrow{\alpha_{\text{sys}}} \langle t + h', S' \rangle \\
& \quad h' = x - h \\
\text{(Log)} & \quad \langle t, [C]_k \bowtie [S]_h \rangle \xrightarrow{\alpha_{\text{lay}}} \langle t + h', [C']_k \bowtie [S']_0 \rangle \\
& \quad k > 0 \\
\text{(Wake)} & \quad \langle t, [C]_k \bowtie [S]_h \rangle \rightarrow_{lazy} \langle t', [C]_k \bowtie [S]_{h'} \rangle
\end{align*}$

Figure 1: The transition relation $\rightarrow_{lazy} \subseteq D \times T \times D$. 

$\text{premise} \rightarrow \text{conclusion}$
Operational Semantics approach [17]; here we informally comment on the rules and we refer the interested reader to [7].

- (Sleep) states that, if at time \( t \) the controller is acting in the proactive mode \( [C]_0 \) and the next observation is scheduled at time \( t + k \), then the controller can idle till that time, hence becoming \( [C]_k \). The label \( \cdot \) of the transition means that this derivation does not involve any action of the target.
- (Mon) describes the behaviour of the controller when it is actively following the execution of its target. As the scheduler prevents the monitoring from entering the idle state, i.e., \( \zeta(C, h) = 0 \), any action of the target started at previous time \( t - h \) and completing at time \( t - h + x \) is monitored. We use a relation characterizing this behaviour of the controller, i.e., \( \implies \).
- (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 \( S \).

The fact that lazy controllers can be reduced to behave as standard security controllers at the semantic level is due to the following theorem, whose proof can be found in [7].

**Theorem 1** ([7]). Let \((\Sigma, C, \implies)\) be a security controller, let \((\Sigma, C, \implies, \rightarrow_{\text{up}}\!, \zeta)\) be the lazy security controller with \( \rightarrow_{\text{up}} \) arbitrarily defined and \( \zeta : C \times T \rightarrow \{0\} \). Then, for any target \( S' \in \mathcal{S} \) and time \( t \in T \)

\[
(t, C \triangleright S) \xrightarrow{\zeta} (t', C' \triangleright S')
\]

if and only if

\[
(t, [C]_0 \triangleright ([S]_0)) \xrightarrow{\ast}_{\text{lazy}} (t', [C']_0 \triangleright ([S']_0))
\]

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

**III. Prototype Implementation and Discussion**

In this section we present our prototype and we discuss on 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 must be monitored.

**Case study:** We imagine a simple medical prescription service infrastructure. The system consists of four actors: (i) a prescription service and (ii) its customers, i.e., doctors, (iii) pharmacies and (iv) a delivery service. Figure 4 depicts the whole system.

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:

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, i.e., pharmacy and deliver, for specifying how the patient accesses 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 customer’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.

The language of policies used by our system is detailed in [7]. Here we only recall that we focus on safety properties which are the class of security policies that can be guaranteed by Schneider’s security monitors [18].

**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\(^1\). 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

\(^1\)Note that here we are not interested in method parameters.
login, add_med, pharmacy, deliver, confirm, cancel

Figure 3: The privacy policy.

The lazy controller is an external application, i.e., running on a different platform with respect to the target service. At each control cycle, the monitor wakes up and requests the current log to the remote platform. Then, the log trace is processed by the policy automaton, see Fig. 3 the check whether a violation occurred. If it the case, the monitor sends a security error signal to the execution platform (here causing the target to be reinitialised). Instead, if the observed trace is legal, the lazy monitor 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 $(p,q), t_{p,q})$ in a two-columns table.

Hibernation times are computed, using the procedure detailed in [7], starting from a description of the target system. Clearly, the system behaviour depends on the customers. We assume that standard behaviour is known, e.g., by analysing the system execution. In our model we considered two possible descriptions: Continuous Time and Discrete Time Markov Chains (CTMC and DTMC, respectively). In particular, we use the following matrices for describing the standard execution of the service.

$$
R = \begin{bmatrix}
0 & 1/30 & 0 \\
1 & 2/5 & 1/4 \\
2 & 0 & 0 \\
\end{bmatrix} \quad P = \begin{bmatrix}
0 & 1/20 & 17/20 & 1/10 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
$$

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

Figure 4: The prescription service scenario.
add_med ($P_{\text{add med}} = 4/5$) and add_hiv ($P_{\text{add hiv}} = 1/20$) transitions.\(^3\)

**Performance evaluation:** The prescription service was developed with Eclipse IDE (Helios Service Release 2) and executed on the 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. 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 5 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 Dijkstra 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. Figure 6 and Figure 7 show the violation detection delays produced by our testing activity.

Note that the delays for CTMC and DTMC-based monitors have completely different meaning and must be interpreted. Indeed, CTMC controllers work under real time settings, i.e., the monitor is created in order to keep 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 executed after a violation. However, it is interesting to compare how the two models behave 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%.

**IV. Conclusion and Related Work**

We presented a prototype implementation of a monitoring environment using lazy controllers for remotely verifying that a target complies with a security policy. Our technique schedules the security checks along with the target 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, in 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. Our analysis highlighted that our method can be customised on target features and system necessities in order to be efficient and to keep under control the risk of security violations.

**Related work:** At the best of our knowledge, the only work developed in a context similar to ours is [20], has been developed concurrently and independently from ours. Roughly, the authors propose a method for estimating the probability that a certain property is respected by an execution given a sample sequence of observations. Interestingly, the problem analysed in [20] is complementary to the one we considered here. Indeed, while their approach aims at providing an estimation of the risk of a violation, we propose a solution for forcing such risk factor to be bounded through a proper scheduling of target observations. We think that our work and the proposal by Stoller et al. represent the first steps toward a novel theory of security monitoring based on lazy, asynchronous observations.

Many authors proposed important contributions to the theory and practice of security controllers, e.g., see [18], [6], [8], [23], [13], [5], [14], [15]. Even though some of these controllers work in timed settings, none of such proposals deal with our working assumptions.

Many solutions, e.g., [8], [23], [14], rely on program transformation or code rewriting. In practice, the security checks are attached to the target code. Security checks can be either instrumented in the control flow of the program or executed by a concurrent agent, i.e., a security monitor. Such methods require a full control on the implementation of the target system. In other words, they follow each step of the guarded execution till they find a violation. Our approach releases this requirement. Indeed, our monitor can be applied to certain systems which are unmodifiable. For instance, our controllers can be applied to the log facility, if one exists, of a remote service for monitoring the side effects of the service execution. In this case, the monitor could observe the log status, changing the frequency of the observations and, possibly, interrupting the service session if the risk of a security violation arises.

The approach presented in [13] proposes a procedure for translating Metric Interval Temporal Logic [3] (MITL) specifications into timed automata [2]. A timed automaton synchronises with its target and checks whether the current execution complies with the given MITL formula. A MITL specification can be applied to a time interval defined over a dense time domain (typically \( \mathbb{R}_+ \)) through proper operators. Note that, also in this case, the monitoring process cannot be suspended and no notion of failure risk is given. Our proposal works under different assumptions and we deem that (given a proper encoding of their semantics) it can be also extended to timed automata.

Some authors propose optimisations of the security controllers based on some static verification techniques. Bartoletti et al. [5] present a language-based approach for the application of security policies having a local scope, namely local policies. Their policies are defined through _usage automata_, i.e., a sort of non-deterministic finite state automata, which can be directly translated into security controllers. Instead of watching the execution with each controller, they model-check the local policies and remove the controllers corresponding to the policies that cannot be violated at runtime. A similar reasoning is also proposed in [10]. Roughly, it presents the notion of _Security-by-Contract_ \((S \times C)\), i.e., a paradigm for driving the deployment, verification and security monitoring of untrusted mobile software. Briefly, if (the contract of) a software satisfies all the security requirements, it can run without restrictions. Otherwise a security controller is applied at runtime. Both these methods are orthogonal to ours. Indeed, program verifications steps are still applicable before the generation of our security controllers in order to avoid unnecessary checks. Also, if a log functionality is available, we do not need to have direct access to the target code, unlike the proposals above.

Beyond security monitoring, many other approaches aim at providing guarantees on remote behaviour. Recently, _trust_ and _reputation_-based systems received mayor attention. For instance, in [16] a model is presented for extending the reputation measures and their computation across different, virtual organisations. Roughly, this system allow different communities, e.g., developers, sellers and customers, to share a common knowledge on the expected reliability of the participants. Differently from our work, these systems often do not bind reputation to actual behaviour of subjects.

In [9] a framework for trust-integrated deployment of mobile software is presented. Briefly, the authors use a trust management system to retrieve trust values that are used to decide whether it is risky to run a certain application. If the program source is considered suspicious, the platform activates a security monitor watching the execution. However, the execution monitor does not consider the specific trust value of its target, i.e., the system uses a “black or white” strategy.

Even though we did not consider explicitly this possibility, there is no reason to exclude that our system can be integrated with similar assumptions. Indeed, the behaviour of lazy controllers can be tuned by changing on the execution parameters, e.g., risk threshold and target behavioural matrices. Intuitively, risk factors can be obtained...
as functions of the execution context, e.g., reputation of the application/platform, and then used to find a desired trade-off between performances and security controls.

REFERENCES


