Aspect-Oriented Adaptation of Access Control Rules

Tomáš Bureš*, Ilias Gerostathopoulos*† Petr Hnětynka*, Stephan Seifermann‡, Maximilian Walter‡, and Robert Heinrich‡
*Charles University in Prague, Faculty of Mathematics and Physics, Czech Republic
Email: {bures, iliasg, hnetynka}@d3s.mff.cuni.cz
†Vrije Universiteit Amsterdam, Department of Computer Science, the Netherlands
‡Karlsruhe Institute of Technology (KIT), Germany
Email: {stephan.seifermann, maximilian.walter, robert.heinrich}@kit.edu

***Abstract*—Cyber-physical systems (CPS) and IoT systems are nowadays commonly designed as self-adaptive, endowing them with the ability to dynamically reconfigure to reflect their changing environment. This adaptation concerns also the security, as one of the most important properties of these systems. Though the state of the art on adaptivity in terms of security related to these systems can often deal well with fully anticipated situations in the environment, it becomes a challenge to deal with situations that are not or only partially anticipated. This uncertainty is however omnipresent in these systems due to humans in the loop, open-endedness and only partial understanding of the processes happening in the environment. In this paper, we partially address this challenge by featuring an approach for tackling access control in face of partially unanticipated situations. We base our solution on special kind of aspects that build on existing access control system and create a second level of adaptation that addresses the partially unanticipated situations by modifying access control rules. The approach is based on our previous work where we have analyzed and classified uncertainty in security and trust in such systems and have outlined the idea of access-control related situational patterns. The aspects that we present in this paper serve as means for application-specific specialization of the situational patterns. We showcase our approach on a simplified but real-life example in the domain of Industry 4.0 that comes from one of our industrial projects.**

***Index Terms*—Self-adaptive systems; security; access control; aspect-oriented.**

I. Introduction

Over recent years, modern smart systems has been influencing more and more parts of every day life. Cyber-physical systems (CPS) and IoT systems like smart traffic, buildings or manufacturing has become ubiquitous. These system are in most cases very dynamic and designed to be self-adaptive. As they closely interact with humans, their security and trust belong to their most important properties [1].

The self-adaptive systems need to cope with many only partially anticipated or even fully unanticipated situations and, as a result, there is an inherent uncertainty in them. Security and trust, which are traditionally modeled rather rigidly, are therefore also needed to cover uncertainty and to be self-adaptable.

In our recent paper [2], we have analyzed and classified uncertainty in security and trust in smart self-adaptive systems from the domain of Industry 4.0. Importantly, we have outlined the idea of access-control related situational patterns, which serve as adaptation strategies in cases when partially anticipated or completely unanticipated situations are encountered in the system. By themselves the patterns were abstract, did not have exact semantic and did not have any realization in implementation.

The goal of this paper is to provide semantics to these patterns by their explicit specification for the domain of Industry 4.0 and realize them using special kind of aspects, whose abstractions and implementation we feature as the main contribution of the paper.

Rather than defining a new specification language, we are using the previously developed DSL [3] for access-control specifications systems. The patterns can be seen as cross-cutting concerns which are applied dynamically over a core access-control specification. The approach is inspired by the aspect-oriented programming [4].

Our solution differs from the traditional aspect-based approaches which define how program is modified via point-cuts (i.e., where program's behavior has to be changed), advices (how program's behavior is to be changed) and additional declaration (structural updates of the program) and which view security as an aspect, but do not provide any special aspect-based abstractions tailored towards security and its adaptation. In our approach we feature special access control adaptation aspects that are used to adapt existing dynamic access-control rules; thus providing a second level of dynamicity into a system with adaptive access-control.

The paper is structured as follows. In Section II we describe the motivation example and patterns defined in [2] and present our approach for capturing access control in dynamic systems. In Section III we show the aspect-oriented definition that provides a concrete semantics of the situational patterns for the domain of Industry 4.0. Section IV presents the implementation of our approach while Section V discusses the related work. Finally, Section VI concludes the paper.

II. Access control

As a particular example, we are using an updated use-case from our previous Trust 4.0 project[1], which targeted the dynamic security in the Industry 4.0 environment, in particular the access control. The use case is a simple one but still it is a realistic scenario created based on descriptions provided by the industrial partners of the project.

In the use case, we assume a factory with multiple workplaces (see Figure 1). Workers in the factory are organized to shifts. Each shift has a foreman and is assigned to a particular workplace. The workers are not allowed to enter another workplace. Additionally, a worker may not enter the workplaces without a head gear that has to be taken from a dispenser. Importantly, the shift assignment is not static and may change every day and thus roles of workers can rapidly change.

In each workplace, there is a machine that can be configured to produce a particular product and the logs of the machine are treated as intellectual property of the factory. The factory is also served by trucks bringing in parts and carrying away manufactured products. Only authorized trucks can enter the factory compound and only in the designated time interval.

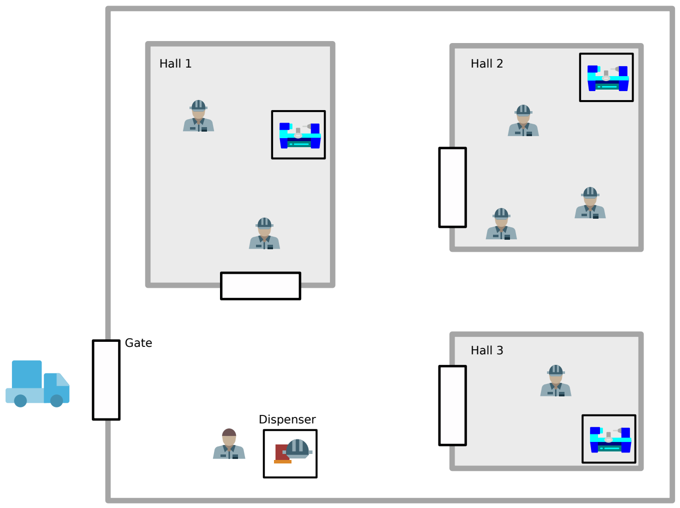

Fig. 1. Factory example

In the scope of the above mentioned project, we have created an approach for specifying access-control in a systems via security ensembles [3]. In the approach, we treat security as self-adaptive architecture, in which an adaptation controller continuously evaluates the conditions (access control rules) defined as ensembles and assigns/removes individual rights to elements in the system. In more detail, the approach is shown in Section II-B.

Note that while our approach is based on the security ensembles, the key abstraction that we pursue in the paper is that the access control is dealt with on two levels: (1) Static access control rules (in form of subject–verb–object) exist in the system. These rules are used for instance for management of physical access via existing solutions utilizing a smart locks with RFID readers. (2) Dynamic situation-specific rules form an adaptation layer above the static rules (#1). The dynamic rules are tied to non-trivial relations between entities and spatio-temporal context. The dynamic rules re-generate (adapt) the static rules based on the current situation.

[1]http://trust40.ipd.kit.edu/home/

While this two-level approach to access control easily captures the dynamic changes in the Industry 4.0 environment and has been successfully evaluated in the project, we have identified a number of situations where the ensemble-based specification by itself is not enough. These situations are connected with potentially rare or unanticipated situations occurring in the system. In [2], we have provided a classification of uncertainty in access control and presented identified situational patterns, which serve as a strategy for dynamic adaptation of security access rules. Namely, we have identified five patterns, which are briefly recapitulated below. These patterns amend the adaptation provided by dynamic situation-specific rules. They provide a second-level of adaptation that adapts the decisions taken by the dynamic rules. While the dynamic rules cover anticipated situations, the patterns handle the partially unanticipated situations.

A. *Situational Patterns*

Pattern 1a—Adding *allow*: A new situation cannot be handled with currently assigned access control rules—a new *allow* rule needs to be assigned, i.e., a new security access rule assigning the *allow* to a component is added to the system. Context of the pattern is as follows. The allow rule is assigned to a component, which either has: (a) such a role in the system that the new rule does not fall outside the component's area of competence, or (b) a similar role in the system as a component that already has the same rule.

Pattern 1b—Adding *deny*: A potentially dangerous situation occurs in the system. The *deny* access control rule is assigned to the component (i.e., a new security access rule assigning the *deny* to a component is added to the system). Context of the pattern is as follows. A component has started to misbehave—accessing more than is usual and/or necessary for it. As a security measure, the *deny* rule is assigned to the component.

Pattern 2a—Removing *allow*: A potentially dangerous situation occurs in the system. The *allow* access control rule is removed from the component (i.e., an existing security access rule assigning the *allow* to a component is removed from the system). Context of the pattern is as follows. A component has started to misbehave and or is broken. As a security measure, the *allow* rule is removed from the component. The pattern is very similar to the *Pattern 1b*—the difference is that the *1b* is used when there is no rule to be removed.

Pattern 2b—Removing *deny*: The system runs in a situation that is blocked by a rule with the *deny* access control rule. The *deny* rule is removed from the component (i.e., an existing security access rule assigning the *deny* to a component is removed from the system). Context of the pattern is as follows. The system can continue in the common operations only if a component can access an entity (e.g., another component) but there is a rule *denying* the access. The rule is removed (the rule can be removed only in the case the rule represents redundancy in the security chain).

Pattern 3 – a new access rule validator: The system runs in a situation that is blocked by a component that validates access for other components (e.g., the component is broken). Another component is chosen as a replacement and serves as a new validator. Context of the pattern is as follows. The selected component has to already have a supervisor-like role in the system (and thus the risk of assigning additional access control rules to it is minimized).

B. *Access control via ensembles*

As mentioned above, we have created an approach for specifying access-control in a system via security ensembles and thus model access control rule assignment as an adaptive system. In this section we very briefly overview the approach; for details see [3].

For rapid creation and experimentation with, we created a Scala-based DSL to define access-control specifications. Listing 1 shows a small part of the specification for the above described example. There are two main concepts—components and ensembles. Components represents physical objects (Door, Gate, Dispenser, Worker) while ensembles define dynamic access control rules over the components. The components are defined via their observable attributes (lines 2–9 in the listing). An ensemble is a dynamically established group of components. Namely, an ensemble determines a particular dynamically emerging situation, identifies components that take part in the situation and specifies access rules for the components. For example the TransportAccessThroughGate (lines 16–24) ensemble assigns a truck the *allow* rule for accessing the factory, but only during the scheduled time period (defined by the situation definition).

As seen from the listing, ensembles can be hierarchically nested (one ensemble defined within another). The top-level ensemble describes a goal of the system as a whole while the nested ensembles represents individual sub-goals. In the example, a single component can be simultaneously in several different situations; thus, it can be a member of several ensembles at the same time.

```
1  class TestScenario() extends Model {
2    object CAS extends Component { /*...*/ }
3    class Gate(/*...*/) extends Component { /*...*/ }
4
5    class Worker(/*...*/) extends Component {
6      val id: String, var position: Position,
7      val capabilities: Set[String], var hasHeadGear: Boolean
8    }
9    /* ... */
10
11   class FactorySystem(factory: Factory) extends RootEnsemble {
12     class GateAccess(gate: Gate) extends Ensemble {
13       initiatedBy(CAS)
14       val assignedTransports = transports.filter(tr =>
             tr.assignedGate == gate)
15
16       class TransportAccessThroughGate(transport: Transport)
             extends Ensemble {
17         situation {
18           (now isEqualOrAfter (transport.scheduledArrival
                 minusMinutes 5)) &&
19             (now isEqualOrBefore (transport.scheduledArrival
                   plusMinutes 15)) &&
20             transport.assignedGate == gate
21         }
22
23         allow(transport, Enter, gate)
24       }
25
26       val transportAccesses = rules(transports.filter(tr =>
             tr.assignedGate == gate).map(tr => new
             TransportAccessThroughGate(tr)))
27     }
28
29     class ShiftTeam(shift: Shift) extends Ensemble {
30       initiatedBy(CAS)
31
32       object AccessToFactory extends Ensemble {
33         situation {
34           (now isEqualOrAfter (shift.startTime minusMinutes
                 45)) &&
35             (now isEqualOrBefore (shift.endTime plusMinutes
                   45))
36         }
37
38         allow(shift.foreman, Enter,
               shift.workPlace.factoryBuilding)
39         allow(assignedWorkers, Enter,
               shift.workPlace.factoryBuilding)
40       }
41
42       object AccessToDispenser extends Ensemble {
43         situation {
44           (now isEqualOrAfter (shift.startTime minusMinutes
                 40)) &&
45             (now isEqualOrBefore shift.endTime)
46         }
47
48         allow(shift.foreman, Use,
               shift.workPlace.factoryBuilding.dispenser)
49         allow(assignedWorkers, Use,
               shift.workPlace.factoryBuilding.dispenser)
50       }
51
52       object AccessToWorkplace extends Ensemble { /* ... */ }
53
54       object AccessToMachine extends Ensemble { /* ... */ }
55
56       object
             NoAccessToMachineSensitiveDataOtherThanFromWorkplace
             extends Ensemble { /* ... */ }
57
58       object AccessToBrokenMachine extends Ensemble {
59         val assignedRepairmen = repairmen.filter(rm =>
               rm.machine == shift.workPlace.machine)
60         allow(assignedRepairmen, Read("logs"),
               shift.workPlace.machine)
61       }
62
63       deny(repairmen, Read("*"), shift.workPlace.machine,
                PrivacyLevel.SENSITIVE)

      /* ... */
      rules(
        AccessToFactory, AccessToDispenser, AccessToWorkplace,
        AccessToMachine, AccessToBrokenMachine,
            CancellationOfWorkersThatAreLate
      )
    }

    val shiftTeams = rules(shifts.filter(shift =>
        shift.workPlace.factoryBuilding.factory ==
        factory).map(shift => new ShiftTeam(shift)))

    val gateAccessRules = rules(gates.map(gate => new
        GateAccess(gate)))
  }

  val factoryTeam = root(new FactorySystem(factory))
}
```

Listing 1. Original security specification

III. Aspect-oriented self-adaptation for access control

The situational patterns presented in the previous section are by themselves abstract and independent of the application domain in which they are applied and the particular access control method used. In this section, we provide precise semantics to the patterns by reifying them into access control adaptation aspects and showcasing them by applying them to our use-case of security ensembles in Industry 4.0 settings.

By specifying the patterns as aspects and not mixing them manually in the primary specification of security ensembles (like the one in Listing 1), we make the primary specification more comprehensible as it focuses on known anticipated situations. The aspects cope with situations that are partially unknown and thus their handling is often more general and applies to multiple situations (i.e., ensembles in our case).

A. Patterns as aspects

A situational pattern (and its instantiation as an access control adaptation aspect) can be seen as a recipe for adaptation of rules in case of some particular type of uncertainty. As such the pattern represents an individual concern that can be applied through the whole specification. Therefore it closely aligns with the idea of *aspects* in aspect-oriented programming (AOP) [4].

AOP targets modularization of a program via separation of cross-cutting concerns. It is done by adding behavior to the code but this additional behavior does not modify the code directly and is specified separately. In more detail, AOP defines a concept of *join point*, which is a point within the program run (e.g., a call of a particular method/function). The *pointcut* is a definition of join points (typically, a single pointcut defines an infinite number of join points). Finally, an *advice* is behavior that has to be executed when the program reaches a specified join point. An *aspect* is the definition of a set of pointcuts and corresponding advices.

In a similar vein, we structure our aspects and adjust them to reflect the structure of the patterns—i.e., the definition (similar to a pointcut) of a situation and the specification of how to modify the assigned access control rules (similar to an advice). In the rest of the section, we present the aspect-oriented specification of the patterns for the example presented in Section II and explain the semantics of our aspect specification.

B. Specification of access control adaptation aspects

The most important parts of the specification are shown in Listing 2. We have designed the specification language as internal DSL in Scala programming languge. This allows for easier integration with existing infrastructures based on Java.

```
object Pattern1a extends AccessControlAdaptationAspect {
  override type SubjectType = Worker
  override type ObjectType = Component

  spec {
    (worker, component) =>

    pointcut {
      existsAllowRule(worker, ActionSelection.ANY, component)
            &&
        component.hasFailure &&
        worker.isForeman
    }

    insert_rules {
      allow(worker, ActionSelection.ALL, component)
    }
  }
}

class Pattern1b extends AccessControlAdaptationAspect {
  override type SubjectType = Worker
  override type ObjectType = Component

  spec {
    (worker, _) =>

    pointcut {
      !existsAllowRule(worker, ActionSelection.ANY,
            ObjectSelection.ANY) &&
         worker.withAnomalousBehavior
    }

    insert_rules {
      deny(worker, ActionSelection.ALL, ObjectSelection.ANY)
    }
  }
}

class Pattern2a extends AccessControlAdaptationAspect {
  override type SubjectType = Worker
  override type ObjectType = Component

  spec {
    (worker, _) =>

    pointcut {
        existsAllowRule(worker, ActionSelection.ANY,
              ObjectSelection.ANY) &&
      worker.withAnomalousBehavior
    }
49
50      delete_rules {
51        allow(worker, ActionSelection.ANY, ObjectSelection.ANY)
52      }
53    }
54  }
55
56  class Pattern2b extends AccessControlAdaptationAspect {
57    override type SubjectType = Worker
58    override type ObjectType = Component
59
60    spec {
61      (worker, component) =>
62
63      pointcut {
64        existsDenyRule(worker, ActionSelection.ANY,
              ObjectSelection.ANY) &&
65          component.hasFailure &&
66          worker.isRepairman &&
67          worker.canRepair(component)
68      }
69
70      delete_rules {
71        deny(worker, ActionSelection.ANY, component)
72      }
73    }
74  }
75
76  class Pattern3 extends AccessControlAdaptationAspect {
77    override type SubjectType = Worker
78    override type ObjectType = Component
79
80    spec {
81      (worker, component) =>
82
83      pointcut {
84        component.getValidator.hasFailure && // e.g. failure of a
              card reader
85          worker.isGuard
86      }
87
88      insert_rules {
89        allow(worker, "validateAccess", component)
90      }
91    }
92  }
```

Listing 2. Aspect-oriented specification

The definition of the aspect-oriented pattern is structured along two components that act as subject and object in the access-control rules derived from security ensembles (i.e., *allow* or *deny* rules in Listing 1). The types of both the subject and the object components need to be provided to allow for applying the pattern via type matching at runtime. The situation when the behavior of the pattern triggers is described within the pointcut declaration, which is a condition expressed over the components and their attributes. The behavior is then either insert_rules or delete_rules (or both) declarations, which prescribe modification of access control rules of the matched components.

In particular, if we take **Pattern 1a** (lines 1–18 in Listing 2), it is defined over a subject of type Worker and an object of type Component. Its pointcut defines a situation that there is a component with failure[2] and there exists some *allow* access control rule between the component and a worker that is a foreman for the particular shift. If the pointcut is evaluated to true for a particular ensemble from the core specification, insert_rules triggers and allows every action for the foreman over the broken component. Namely, if the dispenser of headgear breaks, the foreman will be allowed to fully control the dispenser and thus open it and issue headgear to the shift workers.

In the same manner, the other patterns are instantiated for our example.

C. Detailed semantics

In the previous section we introduced the proposed DSL and described it informally. In this section, we dive deeper and detail the semantics of the individual DSL constructs.

Each access control adaptation aspect is specified as a Scala class that extends the AccessControlAdaptationAspect class. The latter declares two types that should be instantiated by each specific aspect: the SubjectType and the ObjectType. Their values are matched against the types of the subject and object, respectively, of an access control rule (i.e., an *allow*/*deny* rule) of the core specification to determine whether the aspect is applicable at a certain point at runtime. For instance, an aspect that defines its SubjectType to be Worker is only applicable to rules whose subjects are of type Worker or subtypes of it.

For illustration, consider the rule of the core specification (Listing 1, line 48):

```
allow(shift.foreman, Use, shift.workPlace.factoryBuilding.dispenser)
```

The subject of this rule is shift.foreman, its object is shift.workPlace.factoryBuilding.dispenser, and its action is Use. The type of the subject is Worker, while the type of the object is Dispenser, which is subtype of Component. As a result, any access control adaptation aspect with SubjectType equals to Worker and ObjectType equals to Dispenser or Component is applicable to it. In particular, all of the five aspects listed in Listing 2 are applicable in this case.

The second main part of an aspect's specification is the spec construct, which is specified via providing an anonymous function of the form

```
(parameter1, parameter2) => function_body
```

Here, the first parameter is of type SubjectType while the second is of type ObjectType (their names are just variable names that can be used in the function body). Note that, following Scala's syntax, when a parameter is not used in the function body, the underscore character can be used in the spec (e.g., line 25 in Listing 2).

The function body expects two constructs: pointcut and insert_rules or delete_rules. The first is specified via a

[2]We suppose that components report their failure state via their attributes—components like Dispenser or Door has internal monitoring function, while for components like Worker, their failure/misbehaving is detected via external monitoring, e.g., like in [5].

Boolean expression. The pointcut specification can comprise both application-specific clauses(e.g., worker.isForeman or worker.withAnomalousBehaviour) and application-agnostic ones. In the latter case, the constructs existsAllowRule and existsDenyRule are used, which check whether an access control rule of type *allow* or *deny*, respectively, of the core specification is currently active. In using these constructs, the special wildcards ActionSelection.ANY and ObjectSelection.ANY can be used to allow matching to any rule and any object. For illustration, the construct

```
existsAllowRule(worker, ActionSelection.ANY, component)
```

will be positively evaluated at a certain point at runtime when there is an active rule between a subject of the worker's type and object of the component's type, irrespective of the actual action of the rule.

Finally, the construct insert_rules and delete_rules are specified via an *allow* or *deny* rule. This rule is either to be inserted or removed to the system once the aspect is applied at runtime. Here too, the special wildcards ActionSelection.ALL, ActionSelection.ANY and ObjectSelection.ANY can be used—the first one in the specification of an insert_rules construct, while the others in the specification of a delete_rules one. Their semantics are straightforward: they allow matching against either all or any action or object in the set of active rules.

IV. Implementation and evaluation

A. Implementation

As a proof of the concept of the approach presented in the paper, we created an implementation that applies the general approach of the access control adaptation aspects to our use-case of security ensembles in Industry 4.0 settings. Our implementation is based on the TCOOF-Trust framework[3]. TCOOF-Trust uses the ensemble-based access-control specifications defined with our Scala-based DSL (an example is given in Listing 1) and translates them to a constraint solving problem that is solved at runtime by a CSP solver [3]. The answer of the solver is used to determine the components that are assigned to each ensemble, as well as their role in each ensemble—this assignment is in turn used by the framework to assign *allow* and *deny* access control rules to the respective components. To ensure that all access control rules are up to date, the framework continuously runs a single centralized instance of the solver.

Building on our previous work on TCOOF-Trust, we extended the framework to allow for (i) specializing the patterns via aspects for a particular application (this is done by the internal Scala-based DSL exemplified in Listing 2), and (ii) applying the corresponding aspects at runtime. The later allows for manipulating the access control rules derived from the standard ensemble resolution the framework performs.

[3]https://github.com/smartarch/tcoof-trust

The aspect-based access rule manipulation works in the following way. Once an ensemble resolution is performed, the list of *allow* and *deny* rules for the matched components are piped through an aspect processor. The aspect processor tries to match (for each aspect for each access control rule) the type of the aspect's subject and object to the type or super-type of the rule's subject and object. In case of a match, it evaluates the pointcut of the aspect using the matched subject and object. If this is satisfied, then the insert_rules or delete_rules advices are executed. The end result is that, for each aspect application, a set of specific *allow-deny* rules to be added and a set of *allow-deny* rules to be removed is generated.

Since potentially many aspects can be active at the same time and some of them may insert the same rules that others delete, the result of their application depends on the order in which they are applied. In the current implementation, the precedence of aspects is inverse to their specification, i.e., the aspect specified last takes precedence in the application of its rules (which is a simple, static precedence rule).

B. Scalability

To evaluate the proof of the concept, we have applied the extended TCOOF-Trust framework to a Scala-based simulation of the Industry 4.0 use case described in Section II. In the simulation, a number of workers and foremen move in different rooms in the factory going about their shifts. Via triggering different (exceptional) situations such as failures in components that typically act as objects in the access control rules (e.g., doors, equipment, machines) we were able to verify that the different aspects were able to enhance or reduce the permitted actions over the matched components in each situation. We also observed that, for the relatively small number of components (less than 50) that we used in our runs, the performance overhead of applying the aspects by the framework was negligible. This however needs to be further investigated with higher number of components.

The implementation of the aspects on the TCOOF-Trust framework along with the use case implementation are available on Github[4].

V. Related work

To position our work with related approaches, we focus in this related work section on the areas of access control in adaptive systems, aspects in access control, and also aspects in adaptive systems.

Access control in adaptive systems: The classical access control systems are DAC [6] and MAC [7] but they can be used in the simplest cases only. More advanced and still in use is Role-based access control (RBAC) [8], which employs groups to gather access rights for similar users. However, the strict static relationship from groups (with static

[4]https://github.com/smartarch/tcoof-trust-aspects

members) to rules is not suitable for self-adaptive systems with dynamic and unanticipated situations. Another used access control system is Attribute Based Access Control (ABAC) [9], where access is managed over attributes that need to be satisfied for accessing data. An ABAC-based approach is described in [5], which targets dynamic situations in contemporary systems. Nevertheless only user behavior representing a potential attack is considered. An approach in [10] targets access policies creation for dynamically established coalitions, nevertheless, the coalitions are considered only as groups of humans with the same goal and by themselves are not explicitly described. An adaptive access control approach in [11] targets context-based systems, but uses predefined access control policies with predefined exceptions only. The Organisational Based Access Control (OrBAC) extends RBAC by separating the implementation level from the abstract level—an organization here is a group of entities. Coalition-OrBAC [12] extends OrBAC by adding another level for controlling access in dynamically and temporarily created coalitions, however the coalitions themselves are not explicitly described.

Aspects in access control: Aspect-oriented programming is currently a well established approach for increasing modularization of code by allowing clear separation of concerns (even though opinions can be found claiming that AOP can reduce readability of code or even negatively impact modularization of the code—discussions about trade offs can be found, e.g., in [13]). The most common concerns modularized via aspects are typically *logging* and *transaction processing.*

Using aspects for managing access control (or information security in general) is not a new idea and there exist approaches employing them. The systematic literature review in [14] provides an overview of these approaches. All of the reviewed approaches manage access control in software only (e.g., allowing method calls) while our approach primarily deals with physical objects access control. A more important difference is that these approaches treat access control as a single concern (that is separated from an application logic); in our approach we modularize an access control specification itself and each our aspect is a particular part of access control.

An approach closer to physical access control is described in [15] where aspects are used for controlling access to patient medical records. Here, an access control specification can be divided in multiple aspects covering different concerns. Authors also provide aspect templates for common specifications—these templates can be seen as an analogy to our patterns. Nevertheless, similarly to approaches above and contrary to our approach, aspects add access control to a code that is originally without any access control specified.

Aspects in self-adaptive systems: As we treat access control as a self-adaptive architecture, the domain of self-adaptive systems in general is also very related.

One of the first works mentioning employment of aspects in self-adaptive systems is in [16]. Here, advantages of potential usage of aspects are discussed and the paper also proposes an updated MAPE-K loop, where each module is composed of several aspects and these aspects interact with each other via events.

In [17] a middleware for IoT systems (where self-adaptation is usually a mandatory property) is envisioned and it is based on aspect-orient programming principles in order to dynamically reconfigure offered services.

The survey in [18] provides overview of different engineering approaches for self-adaptive systems. Even though the survey discusses a huge number of works (it cites more than 300 works), surprisingly, aspects-oriented approach is directly employed only in two [19], [20] of them—both by the same authors and both a little outdated by now. Here, the approach of aspect-oriented modeling is used, i.e., aspects are used to manage models of systems. In particular, runtime variants of a system are specified as aspect models that can be applied over the core specification of a system when needed at runtime. Aspects are applied as needed—either selected manually, or triggered by a prescribed runtime event. Examples of the aspects are particular internationalization of a system, event filters or permission management. The whole approach is intended as a replacement of low-level configuration files and move them the architectural level. From the high-level point of view, the approach is similar to ours, however it works on the model-level only and, importantly, it allows only reasoning about and modifying the structure, not behavior.

The following works employ aspects in the self-adaptation of system, however primarily only to actually add self-adaptation to a system originally designed without it. In [21], a framework for adaptation of BPEL-based systems is presented. The adaptation is defined and implemented as plugins specified as aspects. A similar approach is in [22], where an extension of BPEL is proposed for monitoring and self-recovery of composite web-services. The extension is also based on aspects.

Another survey in [23] analyzes self-adaptive and secure mechanisms in IoT services. Similarly as above, only a single work that employs aspects has been identified. It is in [24], where aspects are used to add security to a system (i.e., security is a treated as a single concern as in works mentioned above).

From more recent works, the approach used in the Dragonfly simulator [25] is similar to ours. The simulator is intended for testing self-adaptive behavior of drones. The core behavior of a drone is defined directly (states and transitions between them) while via aspects, an exceptional behavior is defined. However, the authors do not provide any patterns or guides to identify this exceptional behavior.

In [26], aspects are used within verification of self-adaptive systems, but only for collecting runtime information about a verified system (i.e., for logging).

VI. Conclusion

In this paper, we have presented an approach for dynamic adaptation of access-control rules for tackling uncertainty in access-control in modern CPS systems. We have demonstrated the application of the approach in the domain of Industry 4.0. The approach is based on access-control adaptation aspects that are used to adapt existing dynamic access-control rules. As such, it is possible to separate the fully anticipated dynamicity in access-control from addressing partial uncertainty. Though we present the approach on the case of our security ensembles, the approach is general—it assumes only that some means of adaptive situation-based access-control (not necessarily via ensembles).

The implementation of the approach is available at https://github.com/smartarch/tcoof-trust-aspects. While applied to the domain of Industry 4.0, there do not seem to be any specificities which should prevent extending it to other domains of CPS.

Acknowledgment

This work has been funded by the DFG (German Research Foundation) – project number 432576552, HE8596/1-1 (FluidTrust), supported by the Czech Science Foundation project 20-24814J, and also partially supported by Charles University institutional funding SVV 260451 and the KASTEL institutional funding.

References

[1] A. Peruma and D. E. Krutz, "Security: a critical quality attribute in self-adaptive systems," in *Proceedings of SEAMS 2018, Gothenburg, Sweden*, 2018, pp. 188–189.

[2] T. Bures, P. Hnetynka, R. Heinrich, S. Seifermann, and M. Walter, "Capturing Dynamicity and Uncertainty in Security and Trust via Situational Patterns," in *Proceedings of ISOLA 2020, Rhodes, Greece*, ser. LNCS, vol. 12477. Springer, 2020, pp. 295–310.

[3] R. Al Ali, T. Bures, P. Hnetynka, J. Matejek, F. Plasil, and J. Vinarek, "Toward autonomically composable and context-dependent access control specification through ensembles," *International Journal on Software Tools for Technology Transfer*, vol. 22, no. 4, pp. 511–522, Mar. 2020.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of ECOOP'97, Jyväskylä, Finland*, Jun. 1997, pp. 220–242.

[5] L. Argento, A. Margheri, F. Paci, V. Sassone, and N. Zannone, "Towards Adaptive Access Control," in *Data and Applications Security and Privacy XXXII*, 2018, vol. 10980, pp. 99–109.

[6] S. D. C. di Vimercati, *Discretionary Access Control Policies (DAC)*. Boston, MA: Springer US, 2011, pp. 356–358.

[7] S. De Capitani di Vimercati and P. Samarati, "Mandatory Access Control Policy (MAC) BT - Encyclopedia of Cryptography and Security," H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, p. 758.

[10] D. Verma, S. Calo, S. Chakraborty, E. Bertino, C. Williams, J. Tucker, and B. Rivera, "Generative policy model for autonomic management," in *Proceedigns of IEEE SmartWorld 2017, San Francisco, USA*, 2017, pp. 1–6.

[8] V. Abreu, A. O. Santin, E. K. Viegas, and M. Stihler, "A multi-domain role activation model," in *Proceedings of of ICC 2017, Paris, France*. Paris, France: IEEE, 2017, pp. 1–6.

[9] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo, "Attribute-Based Access Control," *Computer*, vol. 48, no. 2, pp. 85–88, 2015.

[11] S. Sartoli and A. S. Namin, "Modeling adaptive access control policies using answer set programming," *Journal of Information Security and Applications*, vol. 44, pp. 49–63, 2019.

[12] I. Ben Abdelkrim, A. Baina, C. Feltus, J. Aubert, M. Bellafkih, and D. Khadraoui, "Coalition-OrBAC: An Agent-Based Access Control Model for Dynamic Coalitions," in *Trends and Advances in Information Systems and Technologies*, 2018, vol. 745, pp. 1060–1070.

[13] J. Sánchez and G. T. Leavens, "Reasoning tradeoffs in languages with enhanced modularity features," in *Proceedings of MODULARITY 2016, Malaga, Spain*, Mar. 2016, pp. 13–24.

[14] A. A. Thlunoon and K. Kifayat, "Enforcing Access Control Models in System Applications by Using Aspect-Oriented Programming: A Literature Review," in *Proceedings of DeSE 2017, Paris, France*. IEEE, Jun. 2017, pp. 100–105.

[15] K. Chen, Y.-C. Chang, and D.-W. Wang, "Aspect-oriented design and implementation of adaptable access control for Electronic Medical Records," *International Journal of Medical Informatics*, vol. 79, no. 3, pp. 181–203, Mar. 2010.

[16] R. Haesevoets, E. Truyen, T. Holvoet, and W. Joosen, "Weaving the Fabric of the Control Loop through Aspects," in *Proceedings of SOAR 2009, Cambridge, UK*, Sep. 2010, pp. 38–65.

[17] S. M. Balakrishnan and A. Sangaiah, "Aspect Oriented Middleware for Internet of Things: A State-of-the Art Survey of Service Discovery Approaches," *International Journal of Intelligent Engineering and Systems*, vol. 8, pp. 16–28, Dec. 2015.

[18] C. Krupitzer, M. Breitbach, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems (extended version)," University Mannheim, Working Paper, 2018. [Online]. Available: https://madoc.bib.uni-mannheim.de/44034

[19] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair, "An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability," in *Proceedings of MoDELS 2008, Toulouse, France*, 2008, pp. 782–796.

[20] B. Morin, O. Barais, G. Nain, and J. Jezequel, "Taming Dynamically Adaptive Systems using models and aspects," in *Proceedings of ICSE 2009, Vancouver, Canada*, May 2009, pp. 122–132.

[21] A. Charfi, T. Dinkelaker, and M. Mezini, "A Plug-in Architecture for Self-Adaptive Web Service Compositions," in *Proceedings of ICWS 2009, Los Angeles, USA*, Jul. 2009, pp. 35–42.

[22] T. Huang, G.-Q. Wu, and J. Wei, "Runtime Monitoring CompositeWeb Services Through Stateful Aspect Extension," *Journal of Computer Science and Technology*, vol. 24, no. 2, pp. 294–308, Mar. 2009.

[23] I. Singh and S.-W. Lee, "Self-adaptive and secure mechanism for IoT based multimedia services: a survey," *Multimedia Tools and Applications*, Jan. 2021.

[24] L. Pasquale, M. Salehie, R. Ali, I. Omoronyia, and B. Nuseibeh, "On the role of primary and secondary assets in adaptive security: An application in smart grids," in *Proceedings of SEAMS 2012, Zurich, Switzerland*, Jun. 2012, pp. 165–170.

[25] P. H. Maia, L. Vieira, M. Chagas, Y. Yu, A. Zisman, and B. Nuseibeh, "Dragonfly: a Tool for Simulating Self-Adaptive Drone Behaviours," in *Proceedings of SEAMS 2019, Montreal, Canada*, May 2019, pp. 107–113.

[26] R. Gadelha, L. Vieira, D. Monteiro, F. Vidal, and P. H. Maia, "Scen@rist: an approach for verifying self-adaptive systems using runtime scenarios," *Software Quality Journal*, vol. 28, no. 3, pp. 1303–1345, Sep. 2020.