

Security Checks in Programming Languages for Ubiquitous Environments

Eun-Sun Cho

School of Electrical & Computer Engineering, Chungbuk National University
San 48 Gaeshin-dong Heungduk-gu Cheongju Chungbuk, Korea, 361-763
eschough@cbnu.ac.kr

Kang-Woo Lee

Electronic and Telecommunications Research Institute
161 Kajong-dong Yusong-gu, Taejon, Korea, 305-350
kwlee@etri.re.kr

Abstract

Recently, ubiquitous systems gain lots of attention from many researchers because they provide many features that appeal to their users, such as context-awareness, intelligence and user mobility. However, to fulfill these features, these systems collect and use vast amounts of personal information. Therefore, privacy protection becomes a key issue in the ubiquitous systems.

This paper proposes a security policy description model for a ubiquitous language, which supports universal quantification that is essential for privacy description. It also presents a static checker to extract the rules that will be possibly fired under a given credential and a policy. This work is based on our ubiquitous language PLUE(a Programming Language for Ubiquitous Environment).

1 Introduction

Since access control is essential in data management, there have been valuable results in database security including those on XML data. As knowledge base management and ubiquitous computing are becoming a hot issue, security in ontological data gets focuses from many researchers.

However, few of them deals with the security holes with the programs managing data, which may bring out problems in real worlds. For example, SQL injection vulnerabilities [1] may allow a remote attacker to gain access to the database.

We propose a first step to reduce security flaws in ubiquitous programs based on access control mechanism. To help programmers build secure ubiquitous application easily, we suggest an object-oriented and rule-base language with a security policy description facility. In addition, we propose a

static analyzer to determine required rights for evaluation of each rule,

Next section overviews PLUE, a Programming language for Ubiquitous Environment. Section 3 introduces access control policy for PLUE data model. Section 4 suggests a static analyzer which extracts the information about the required access rights for each rule. Section 5 covers related works and Section 6 concludes the paper.

2 A Quick Introduction to PLUE

PLUE is a programming language for writing ubiquitous applications. It is basically an extension of Java programming language, and in fact, its compiler is a pre-processor of the Java compiler. The key extensions in PLUE are remote object invocations, common data model for context, and ECA rule processing. We consider that they are essential to the ubiquitous programming language. In this section, due to space limitation, we just briefly explain the only features that are relevant to the security.

The following is an example PLUE program that implements a smart-room application with which we are going to explain the key features of PLUE.

2.1 Distributed Object Invocation

Executions of a ubiquitous application are inherently distributed therefore, the remote invocation is a crucial feature in PLUE PLUE exploits CORBA to support remote object invocation, and enables programmers to invoke remote CORBA services transparently.

In the above example, ‘\$place.light’ and ‘\$place.air_conditioner’ refer to CORBA objects, by which the SmartRoom application is able to

```

task SmartRoom {
  on ( $place.UserEntered e ) {
    $place.light.turnOn();
  }

  on ( $place.UserEntered e ) {
    if ( $place.temperature
        > e.user.preferred_temp.high )
      $place.air_conditioner.turnOn();
  }
}

```

control the light and air-conditioner in the room pointed by \$place.

2.2 Context Model

One of key characteristics of ubiquitous computing is context-awareness. Ubiquitous applications, to be context-aware, need refer to context information. Thus, most ubiquitous systems address data model for context information[8]. PLUE provides application developers with a context data model, called Universal Data Model (UDM). The UDM looks similar to OEM[9, 2] worked by Stanford University, which denotes data as an edge-labeled graph. Using the data model, application developer should define all the context information including places, user profiles and preferences, and application-specific data.

Access to data in the context database is based path expression starting from given entry nodes. PLUE provides three entry nodes: \$place, \$task, and \$owner.

\$task points to the root node from which application-specific data are described. \$owner refers to the node from which the owner's information is represented, and \$place implies the entry node of the place where the owner is placed. As user moves around many places, the \$place points to the corresponding entry node of the place.

Using the path expressions, starting from the designated entry nodes, a PLUE application is able to reach all the necessary context information, such as the application-specific data, the person who executes this application, and the place where the owner is currently placed.

As shown in the above example, \$place.temperature and \$place.air_conditioner points to the CORBA objects that deal with the thermometer and the air conditioner in the place where the person who executes this application is placed, respectively. Service invocation on a path is also possible, for instance \$place.airconditioner.turnOn().

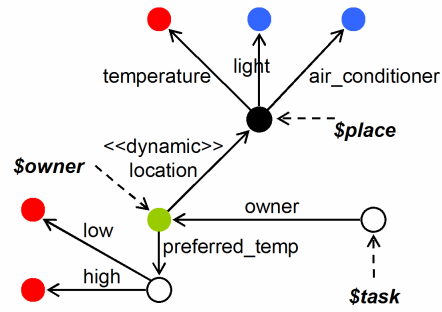


Figure 1. Context Data for the SmartRoom Application

```

task          = task task_name { rules }
rules        = rules
              | on (path e) stmt
path         = entry_points
              | path.x
entry_points = $place | $task | $owner.

```

Figure 2. Grammar: Task and Rules

2.3 ECA Rule Processing

Rule-based programming is another key extension in PLUE. Ubiquitous applications are usually expected to be intelligent enough to provide services to their users proactively and invisibly. They intend to replace domain expertise and perform some work with little human intervention. Rule-based programming is known to satisfy these kinds of requirements. In PLUE, programmers are able to write rules for their applications. PLUE compiler translates these rules into the Jess[7] rules and generates Java codes that call Jess rule engine to process the rules.

Rules in PLUE are augmented with events so that the rules are fired whenever the events are received. Usually these rules are called Event-Condition-Action (ECA) rules and intensively used in the domain where applications are required to react to the changes in the contextual environment. As shown in above example, ECA rules are written in 'on (event-expression) statements_block' construct. In this example, the light in a place will be turned on when a person enters the place, and the air conditioner will be turned on when the temperature in the place becomes higher than the maximum endurable degree.

Fig. 2 shows the grammar for tasks and rules.

head and path are different in that a path represents a data from the ubiquitous knowledge base rooted by a

```

stmt ::= x
      | head.x
      | path.x
      | x := e
      | head.x := e
      | path.x := e
      | new c
      | e ; e
      | this
      | if e then e else e
      | e.m(e)
head ::= x | head.x
: qc  class name
m     method name
x     variable name

```

Figure 3. Syntax of the language

entry_points while a *head* is a usual object path in a program. *stmt* means statements as in any programming languages including Java.

We consider a core of Java shown in Fig. 3 for presentation simplicity as follows; We assume one formal parameter for each method, one field variable for each object and no arrays. We do not consider loops, which can be simulated by recursions. Neither do we consider local variables since they have no important effects on our analysis.

3 Access Control Policy Description

An access control policy in PLUE is a list of $\langle \text{Credentials}, \text{Objects}, \text{Rights} \rangle$, where *Credentials* represents a set of the subject to do the job, *Objects* is for a set of the targets on which the subject do an action, and *Rights* is a set of the actions which are allowed by a specific subject on a target object. Each item in a policy for PLUE has a single credential, a single object and multiple rights. Currently we consider only browse (B) and modify (M) for the access rights. The default is *not-allowed*. To change the default, tuples with ANY can be used, that is, $\langle \text{ANY}, \text{ANY}, B/M \rangle$

We introduce the policy item with the universal quantification which are essential for privacy – for instance, ‘for all *X*, *X* is allowed to access its own data’. Actually, PLUE does not the explicit universal quantifier; a hidden ‘ \forall ’ is assumed before each variable in the policy definition, as shown in Fig. 4.

Here is the sample policy for above the example. Anyone can turn on the light, while one’s preferred temperature data can be accessed only by her.

```
h in House (ANY, h.light, B/M)
```

```

policy p ::= policy
           |[var1[∈ template1], ...varn[∈ templaten]]
           <c, o, t> ∈ Credentials × Objects × Rights
           where templatei ∈ Templates
c       ::= chead | ANY
chead   ::= vari | chead.x | chead[in o].x | chead[== o].x
o       ::= ohead | ANY
ohead   ::= vari | o.x | o[in c].x | c[== c].x
t       ::= B | M | B/M

```

Figure 4. Access Control Policy Grammar

```

p in Person (p, p.preferred_temp.high, B/M)
p in Person (p, p.preferred_temp.low, B/M)
...

```

The semantics of the policy has intuitive meaning that $c \in \text{Credentials}$ is allowed to do $t \in \text{Rights}$ on $o \in \text{Objects}$.

4 A Static Analysis for Rules Firing

4.1 An abstract interpreter

As shown above, due to the separate security policy, PLUE programs do not have to be massed up with access control logic. However, there still remains complication in security-related programming; the behaviors of programs are not clear in the development phase, since the access control violation may or may not prohibit firing its rules. This paper introduces our elaboration on a static detector telling the programmers ‘which rules are expected to be evaluated by which credentials’ to help debugging. This approach prevents the unnecessary rollbacks that might be caused by access control violation.

The concrete semantics of the PLUE action which the static analyzer depends on is omitted by the lack of space. As the task program is running, it gathers the required access right information into a global state.

Fig. 6 shows the abstract interpreter for the static analysis. *InitField()* means the initial value for a field in an instance of class *C*. The \cup operator is for union of sets and field-wise union on relations.

The next theorem mentions the correctness of the abstraction.

Theorem 1 For any expression *expr*,

$$\text{Abspost-state} \circ \text{fix } \mathcal{F}[\![\text{expr}]\!] \sqsubseteq \text{fix } \mathcal{F}^\#[\![\text{expr}]\!] \circ \text{Abspre-state}$$

, when *Abspre-state* is the abstraction function:
 $(\text{Env}_\perp \times \text{Heap}_\perp \times \text{VarRights} \times \text{AllRights}) \rightarrow (\text{ORef} \times$

object reference	$r^\# \in ORef_\perp^\# = \text{Labels for new expressions}$
heap	$h^\# \in Heap_\perp^\# = ORef^\# \rightarrow ORef_\perp^\#$
environment	$\sigma^\# \in Env_\perp^\# = 2^{ORef_\perp^\#} \times 2^{ORef_\perp^\#}$
required rights (for expr)	$q^\# \in ExpRights^\# = 2^{Objects \times Rights}$
required rights (for var)	$z^\# \in VarRights^\# = 2^{Objects \times Rights} \times 2^{Objects \times Rights}$
all required rights	$a^\# \in AllRights^\# = 2^{Objects \times Rights}$

Figure 5. Domains of abstract interpreter

$ExpRights \times Env_\perp \times Heap_\perp \times AllRights \times VarRights$), and $Abs_{post-state}$ is the abstraction function: $(ORef \times ExpRights \times Env_\perp \times Heap_\perp \times AllRights \times VarRights) \rightarrow (ORef^\# \times ExpRights^\# \times Env_\perp^\# \times Heap_\perp^\# \times AllRights^\# \times VarRights^\#)$

The abstraction functions and the proof of the theorem are omitted.

4.2 Extracting useful information

The $AllRights$ information a in the result of the abstract interpreter will be used to extract information on which rules will be evaluated under a specific circumstance. To identify the rules that will be fired without access control violation, we first define the meaning of "being expected to be fired".

Definition 1 A rule R is expected to be fired with a credential c , if and only if c has all rights to browse and/or modify the data required for execution of the rule action.

A rule can be fired by a given credential c , when the policy p describes that c has enough rights that subsumes what a has.

Theorem 2 A given credential c and a given policy p , the rule R is expected to be fired, which is denoted by $ExF(R, c, p)$, if and only if,

$$S_{ar}[a] \subseteq \bigcup_{\langle c, o, t \rangle \in p} S_p[\langle o, t \rangle]$$

where $AllRights$ a is derived from the abstract interpretation on R , and S_{ar} means semantic function for $AllRights$, while S_p is the semantic function for Policy.

More useful information can be extracted for ubiquitous application developers. Following theorems show two helpful properties.

Theorem 3 For a given credential c , a set $\{R | ExF(R, c, p)\}$ extracts all the rules expected to be fired according to the policy p .

Theorem 4 For a given rule R , a set $\{c | ExF(R, c, p)\}$ extracts all the credentials that is expected to fire the rule R .

5 Related Works

There are various works on the ubiquitous security service. [3, 5] suggests a middleware which enables access control policy description to define users' view. But they elaborated only on the data model without consideration of application development. Our work is tightly bound to the ubiquitous programming language PLUE by providing a static analyzer for the language which will help application programmers.

Previous works on XML access control could be mapped into ubiquitous data models. However, previous well-developed XML-based access control models [6], can not be easily adopted for PLUE due to the following points;

- XML based access controls do not consider service invocation.
- Neither do they consider the quantification on credentials and objects. which is necessary for describe the privacy.

In this paper, we mainly focus on these issues; neither object granularity issues nor delegation policies are covered [6].

Although there are also various works in information flow[10, 11, 4] with multi-level security support, we could not find out so far the result on both programming constructs and the access control over composite paths with special root nodes as in ubiquitous environments.

6 Conclusions

This paper suggests a security policy model for ubiquitous programs, which allows universal quantification that is essential for privacy description. It also provides a static checker extracting the rules that be possibly fired under a given credential and a policy. The checking mechanism is designed for our event-based ubiquitous language, PLUE. The checker can also be extended to get other useful information, for example, the set of possible credentials that will fire a specific rule with a given policy. What we consider our future works are;

$$\begin{aligned}
\mathcal{E}^\# : Expr &\rightarrow (Env_\perp^\# \times Heap_\perp \times VarRights \times AllRights) \rightarrow (ORef^\# \times ExpRights \times Env_\perp^\# \times Heap_\perp^\# \times AllRights \times VarRights) \\
\mathcal{F}^\# : (Expr &\rightarrow (Env_\perp^\# \times Heap_\perp \times VarRights \times AllRights) \rightarrow (ORef^\# \times ExpRights \times Env_\perp^\# \times Heap_\perp^\# \times AllRights \times VarRights)) \\
&\rightarrow (Expr \rightarrow (Env_\perp^\# \times Heap_\perp \times VarRights \times AllRights) \rightarrow (ORef^\# \times ExpRights \times Env_\perp^\# \times Heap_\perp^\# \times AllRights \times VarRights)) \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket x \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \langle \sigma^\#(x), z^\#(x), \sigma^\#, h^\#, a^\# \cup z^\#x, z^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket \text{this} \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \langle \sigma^\#(\text{this}), z^\#(\text{this}), \sigma^\#, h^\#, a^\# \cup z^\#(\text{this}), z^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket \text{head}.x \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \langle rs_1^\#, k_1^\#, \sigma^\#, h^\#, a_1^\#, z_1^\# \rangle = \mathcal{E}^\# \llbracket \text{head} \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle \\
&\quad k_1^\# = k_1^\# \cup \{(p.x, B) \mid p \in k_1^\#\} \\
&\quad \text{in } \langle \{h^\#(r_1^\#) \mid r_1^\# \in rs_1^\#\}, k_1^\#, \sigma^\#, h^\#, a_1^\# \cup k_1^\#, z^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket \text{path}.x \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \langle rs_1^\#, k_1^\#, \sigma^\#, h^\#, a_1^\#, z_1^\# \rangle = \mathcal{E}^\# \llbracket \text{head} \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle \\
&\quad \text{in } \langle \{h^\#(r_1^\#) \mid r_1^\# \in rs_1^\#\}, k_1^\# \cup \{(path.x, B)\}, \sigma^\#, h^\#, \\
&\quad \quad a_1^\# \cup k_1^\# \cup \{(path.x, B)\}, z^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket x:=e \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \langle rs_1^\#, k_1^\#, \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle = \mathcal{E}^\# \llbracket e \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle \\
&\quad \text{in } \langle r_1^\#, k_1^\#, \sigma_1^\#[\sigma_1^\#(x) \cup \{r_1^\#\}^\# / x], h_1^\#, a_1^\# \cup k_1^\#, z_1^\#[k_1^\# / x] \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket \text{head}.x:=e \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \langle rs_1^\#, k_1^\#, \sigma^\#, h^\#, a_1^\#, z_1^\# \rangle = \mathcal{E}^\# \llbracket \text{head} \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle \\
&\quad \langle rs_2^\#, k_2^\#, \sigma_2^\#, h_2^\#, a_2^\#, z_2^\# \rangle = \mathcal{E}^\# \llbracket e \rrbracket \langle \sigma^\#, h^\#, a_1^\#, z_1^\# \rangle \\
&\quad k_1^\# = k_1^\# \cup k_2^\# \cup \{(p.x, M) \mid p \in k_1^\#\} \\
&\quad h_2^\# = \bigcup_{r_1^\# \in rs_1^\#, r_2^\# \in rs_2^\#} h_2^\#[(h_2^\#(r_1^\#) \cup \{r_2^\#\}) / r_1^\#] \\
&\quad \text{in } \langle rs_2^\#, k_1^\#, \sigma_2^\#, h_2^\#, a_2^\# \cup k_1^\#, z_2^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket \text{new } c \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \beta^\# \text{ a new id for each new expression} \\
&\quad \text{in } \langle \{\beta\}, k^\#, \sigma^\#, h^\#[\text{InitField}(c) / r^\#], a^\#, z^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket e_1.m(e_2) \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \langle rs_1^\#, k_1^\#, \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle = \mathcal{E}^\# \llbracket e_1 \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle \\
&\quad \langle rs_2^\#, k_2^\#, \sigma_2^\#, h_2^\#, a_2^\#, z_2^\# \rangle = \mathcal{E}^\# \llbracket e_2 \rrbracket \langle \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle \\
&\quad \langle rs_3^\#, k_3^\#, \sigma_3^\#, h_3^\#, a_3^\#, z_3^\# \rangle = \bigcup_{\lambda, m, x, e \in \{\text{Method}(r^\#, m) \mid r_1^\# \in rs_1^\#\}} \\
&\quad \quad \mathcal{E}^\# \llbracket e \rrbracket \langle rs_1^\#, rs_2^\#, h_2^\#, a_2^\#, z_2^\#[k_1^\# / \text{this}][k_2^\# / x] \rangle \\
&\quad \text{in } \langle rs_3^\#, k_3^\#, \sigma_3^\#, h_3^\#, a_3^\#, z_3^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket e_1; e_2 \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \langle rs_1^\#, k_1^\#, \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle = \mathcal{E}^\# \llbracket e_1 \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle \\
&\quad \langle rs_2^\#, k_2^\#, \sigma_2^\#, h_2^\#, a_2^\#, z_2^\# \rangle = \mathcal{E}^\# \llbracket e_2 \rrbracket \langle \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle \\
&\quad \text{in } \langle rs_2^\#, k_2^\#, \sigma_2^\#, h_2^\#, a_2^\# \cup k_1^\# \cup k_2^\#, z_2^\# \rangle \\
\mathcal{F}^\# \mathcal{E}^\# \llbracket \text{if } e_1 \ e_2 \ e_3 \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle &= \text{let } \langle rs_1^\#, k_1^\#, \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle = \mathcal{E}^\# \llbracket e_1 \rrbracket \langle \sigma^\#, h^\#, a^\#, z^\# \rangle \\
&\quad \text{in } \mathcal{E}^\# \llbracket e_2 \rrbracket \langle \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle \cup \mathcal{E}^\# \llbracket e_3 \rrbracket \langle \sigma_1^\#, h_1^\#, a_1^\#, z_1^\# \rangle
\end{aligned}$$

Figure 6. Definition of abstract interpreter

- reflecting granularity and subsuming relationship in the policy model
- refining the concrete semantics
- introducing transaction and composite event, and re-defining the security policy description.

References

- [1] Sql injection attacks - are you safe? <http://www.sitepoint.com/article/794>.
- [2] Serge Abiteboul, Peter Buneman, and Dan Susiu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.
- [3] Lalana Kagal and Tim Finin and Anupam Joshi. Moving from security to distributed trust in ubiquitous computing environments. *IEEE Computer*, December 2001.
- [4] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a java-like language. In *16th IEEE Computer Security Foundations Workshop (CSFW-16)*, July 2003.
- [5] V. Benzaken, M. Burelle, and G. Castagna. Information flow security for xml transformations. In *Proc. of Asian Computing Science Conference (ASIAN'03)*, 2003.
- [6] Elisa Bertino and Elena Ferrari. Secure and selective dissemination of xml documents. *ACM Trans. Inf. Syst. Secur.*, 5(3):290–331, 2002.
- [7] Ernest Friedman-Hill. *Jess in Action*. Manning, 2003.
- [8] K. Henriksen, J. Indulska, and A. Rakotonirainy. Modeling context information in pervasive computing systems. In *1st International Conference on Pervasive Computing*, 2002.
- [9] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), September 1997.
- [10] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, Texas, January 1999.
- [11] Francois Pottier, Christian Skalka, and Scott F. Smith. A systematic approach to static access control. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 30–45. Springer-Verlag, 2001.