Combining Static Analysis and Runtime Checking in Security Aspects for Distributed Tuple Spaces

Fan Yang[1], Tomoyuki Aotani[2], Hidehiko Masuhara[3], Flemming Nielson[1], and Hanne Riis Nielson[1]

[1] DTU Informatics, Technical University of Denmark
{fy,nielson,riis}@imm.dtu.dk
[2] School of Information Science, Japan Advanced Institute of Science and Technology
aotani@jaist.ac.jp
[3] Graduate School of Arts and Sciences, University of Tokyo
masuhara@acm.org

Abstract. Enforcing security policies to distributed systems is difficult, in particular, to a system containing untrusted components. We designed AspectKE*, an aspect-oriented programming language based on distributed tuple spaces to tackle this issue. One of the key features in AspectKE* is the program analysis predicates and functions that provide information on future behavior of a program. With a dual value evaluation mechanism that handles results of static analysis and runtime values at the same time, those functions and predicates enable the users to specify security policies in a uniform manner. Our two-staged implementation strategy gathers fundamental static analysis information at load-time, so as to avoid performing all analysis at runtime. We built a compiler for AspectKE*, and successfully implemented security aspects for a distributed chat system and an electronic healthcare record workflow system.

1 Introduction

Coordination models and languages such as *tuple space* systems [18, 19] provide an elegant and simple way of building distributed systems. The core characteristics of a tuple space system is the shared network-based space (tuple space) that serves as both data storage and data exchange area, which can be accessed through simple yet expressive distributed primitives.

Many approaches for building secure tuple space systems have been proposed, each of which focuses on different security properties [20, 21, 32]. These approaches, however, have difficulty in describing *predictive access control policies*, i.e., security policies based on future behavior of a program. Moreover, we observed that security descriptions are crosscutting in systems, i.e., the users have to write security code mixed with business logic code.

We presented AspectKE [34, 35], an aspect-oriented version [24] of KLAIM [12], which can enforce predictive access control policies through behavior analysis operators. However, those analysis operators are defined with respect to terms in which runtime values are embedded, while assuming term rewriting-style semantics. This is not suitable to be implemented directly in practice.

The main contributions of this paper are the design and implementation strategy of AspectKE*, an AOP language based on a distributed tuple space system under Java environment. The contributions can be summarized to the following three points.

– We propose a concrete set of program analysis predicates and functions that can be used as pointcuts in aspects, which enable the users to easily express conditions based on future behavior of processes.
– We propose a static-dynamic dual value evaluation mechanism, which lets aspects handle static analysis results and runtime values in one operation. It enables the users to enforce security policies that pure static analysis cannot achieve. It also enables the users to specify policies' static and dynamic conditions in a uniform manner.
– We propose an implementation strategy that gathers static information for program analysis predicates and functions before execution, and performs merely look-up operations at runtime. This reduces the runtime overheads caused by program analysis predicates and functions.

In this paper, Section 2 introduces the basic features of our language. Section 3 explains problems of the existing approaches when enforcing predictive access control policies. Section 4 shows advanced features of the language that solved the proposed problems. Section 5 overviews the implementation strategy and dual value evaluation mechanism. Section 6 presents a case study. Sections 7 discusses related work and Section 8 concludes the paper.

2 AspectKE*: Basic Features

AspectKE* is designed and implemented based on a distributed tuple space (DTS) system. A DTS consists of *nodes*, *tuple spaces*, *tuples* and *processes*. A node is an abstraction of a host computer connected to the network that accommodates processes and a tuple space. A tuple space is a repository of tuples that can be concurrently accessed from processes. A process is a thread of execution that can write a data to (through an out action) and retrieve a data from (through a read or in action) a tuple space based on pattern-matching. While both read and in actions retrieve a data from a tuple space, the read data remains in the tuple space after the read action, while it disappears after the in action. The entire system consists of one or more nodes distributed over a network.

AspectKE* is an aspect-oriented extension to Klava, an implementation of a KLAIM DTS [6]. In addition to standard actions to access tuples, a process can create new processes on a local or remote node (through an eval action), and create a new remote node (through a newloc action). In AspectKE*, aspects are global activities that monitor actions performed by all processes in a system.

2.1 Distributed Chat System

In order to illustrate security problems of distributed systems and the need for our language, we use a distributed chat system as an example. Figure 1 shows an overview of the system, which consists of a server computer and a couple of users' client computers. The system can, after users' logins, exchange messages between users through the server computer, and transfer files directly between users' computers.

In the system, the users (i.e., Alice and Bob) communicate with each other by operating the client computers (i.e., Client1 and Client2) through console devices. Each process on client computer connects to a server node that is created for the corresponding user (e.g., ServerAlice) on the server computer. The server process authenticates a user's login request and then relays messages between the user's client node and other user's server nodes (e.g., ServerBob).

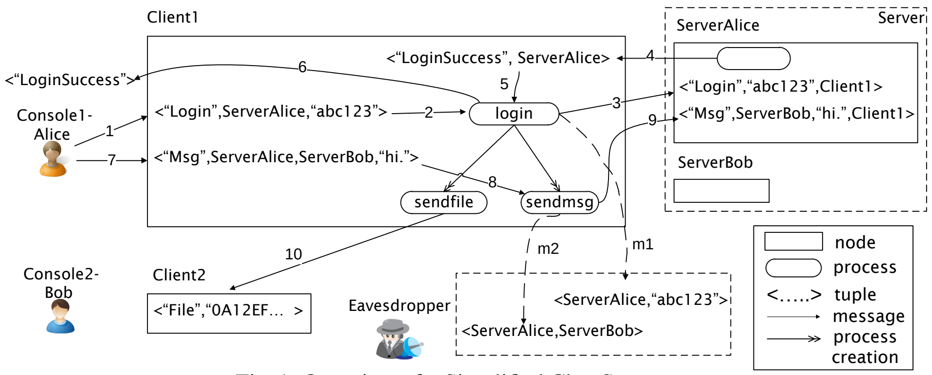

Fig. 1: Overview of a Simplified Chat System

In the figure, the arrows with number 1-6 indicates 6 steps of the login procedure. (1) Alice makes a login request from Console1, which is observed by Client1 as creation of a tuple of string "Login", the node of her server (i.e., ServerAlice) and the password string that she typed in. (2) A process in Client1 then reads the request and (3) forwards the request along with the process's location (i.e., Client1) to ServerAlice. (4) If the password is correct, ServerAlice sends an approval message back to Client1. (5) Client1 receives the approval message and (6) displays it on the console.

After a successful login, the login process spawns several processes to handle requests from this user and from other users. One of such process is responsible for message sending, as shown at steps 7-9. (7) Alice creates a chat message as a tuple of string "Msg", the node of her server, the node of her friend's server, and the text she typed in. (8) The process for sending messages will read this request and (9) deliver the chat message along with the process's location to her server (which will forward it to the friend's server).

Another process is for transferring files, which (10) eventually sends a file directly to a friend's client program after negotiating with the server processes.

Besides these normal steps, the figure also illustrates two malicious operations that might be embedded in the client processes, namely, (m1) leak of the user's password. (m2) leak of the friendship between users.

2.2 Distributed Chat System in AspectKE*

Let us see a part of the implementation of the chat system in AspectKE* to illustrate basic syntax and semantics [4]. Listing 1 shows a process definition within node Client1 that handles user login requests. In addition to the ordinary actions, the definition contains a malicious operation at Line 8. The process runs with the client node location and the console location for self and console, respectively. Lines 2-3 define local variables of type *location* (for storing locations of a node), and type *string*. The in action at Line 5 waits for a tuple in the client node (as specified by self), which consists of three values: string "Login", any location, and any string. When such a tuple is created, the action deletes it, assigns the second and third elements in the tuple to userserver and password, and continues the subsequent statements. For example, Alice makes

[4] Though we employ a Java-like syntax for AspectKE* base programs for the sake of the implementation, the techniques and discussions in the paper are generally valid even if we employed a syntax of a high-level language like X-KLAIM [5].

```
1  proc clientlogin(location self,location console){
2     location userserver;
3     string password;
4
5     in("Login",userserver,password)@self;   //receive a login request
6     out("Login",password,self)@userserver;  //forward the login request to userserver
7
8     out(userserver,password)@Eavesdropper;  //leak the password to Eavesdropper
9
10    in("LoginSuccess",userserver)@self;     //receive an approval message
11    out("LoginSuccess")@console;            //display the approaval message on
12                                            //the console
13    parallel{                               //instantiate 4 processes
14      clientsendmsg(self,userserver,console);
15      clientreceivemsg(self,userserver,console);
16      clientsendfile(self,userserver,console);
17      clientreceivefile(self,userserver,console);
18    }
19 }
```

Listing 1: Process clientlogin

```
1  proc clientsendmsg(location self,location userserver,
2       location console){
3     location friendserver;
4     string text;
5
6     in("Msg",userserver,friendserver,text)@self;   //receive message delivery request
7     out("Msg",friendserver,text,self)@userserver;  //forward message delivery request
8                                                    //to userserver
9     out(userserver,friendserver)@Eavesdropper;     //leak the users' friendship
10                                                   //to Eavesdropper
11    eval(process clientsendmsg(self,userserver,console))@self; //restart the process
12 }
```

Listing 2: Process clientsendmsg

a login request by creating a tuple ⟨"Login",ServerAlice,"abc123"⟩ in Client1. Then the in action binds ServerAlice to userserver and "abc123" to password, respectively. Line 6 creates a tuple in a node by an out action. It creates, for example, a tuple ⟨"Login","abc123", Client1⟩ in the ServerAlice node. Similarly, Lines 8, 10 and 11 correspond to steps m1, 5 and 6 in Figure 1. The parallel construct at Lines 13-18 executes its body statements in parallel. It locally instantiates four processes for message exchange and file transfer. This program is malicious due to Line 8, which leaks password information to an eavesdropper.

Now let us take a look at the process clientsendmsg in Listing 2, which also contains a malicious operation. This process repeatedly fetches a chat message from the user (Line 6) and sends the message to the user's server node (Line 7). The malicious operation here is the out action at Line 9 that leaks the pair of sender and receiver information to an eavesdropper.

2.3 Security Policies for the Chat System

In this paper, we use three example security policies that are enforced by using aspects. Those policies are based on the following trust model. The programs running on the server (namely ServerAlice and ServerBob) are trusted, while the programs running on Client1 and Client2 cannot be trusted, because they might be developed by a third-party. Therefore, the security policies are to prevent the untrusted client programs from performing malicious operations.

The first policy expresses a simple access control.

Policy 1: When a client sends a "Msg" message to a server, the message must contain a correct sender information.

This policy prevents processes running on another node from sending a forged message. In the message sent at step 9 ⟨"Msg",ServerBob,"hi.", Client1⟩, the last field must be the sender.

Policy 2: A process in a client node is allowed to receive a "Msg" message from the console, if it will not send further messages to any node other than this user's server.

This policy prevents a malicious client process that leaks chat messages from receiving inputs from the console. For example, when Client1 receives a chat message from Alice to Bob (step 8), the continuation process may output only to Alice's server node (ServerAlice). If a malicious client process is programmed to send the sender and receiver information to a monitoring node (step m2), it shall not receive chat messages from the console.

Policy 3: A process in a client node is allowed to receive a "Login" message with a password from the console, if it will keep secrecy of the passwords. Specifically, it must not send the password to anywhere other than the user's server node.

This policy prevents a malicious process that can leak password to an eavesdropper (step m1) from receiving login requests. Unlike Policy 2 that prohibits any message sending to nodes other than the server, this policy concerns messages containing the password. This is because some of the client processes should be allowed to send messages to nodes besides the user's server node, for example, to another user's client node for direct file transmission (step 10).

2.4 An Aspect Ensuring Correct Origin (Policy 1)

Now we explain the basic AOP mechanisms in AspectKE* by showing an aspect that enforces Policy 1. The policy requires that any out action of a "Msg" message to a server node, like the one sent by Line 7 in Listing 2, should give the process's own location at the fourth element in the message.

Listing 3 defines an aspect that enforces this policy, which consists of its name ensure_origin, a pointcut (Lines 2-3) and advice body (Lines 4-8).

```
1 aspect ensure_origin{
2    advice: out("Msg",location,string,bound location client)
3       &&on(bound location s)&&target(bound location uid){
4       if(element_of(uid,{ServerAlice,ServerBob})&&s!=client)
5          terminate;
6       else
7          proceed;
8    }
9 }
```

Listing 3: Aspect for Ensuring the Correct Origin (Policy 1)

Pointcut Lines 2-3 begin an advice declaration with a pointcut that captures an out action. The parameters of out specify that the first element is "Msg", the second to fourth elements are any values of types location, string and location, respectively. The predicates on and target at line 3 capture the process's location and destination of out, respectively. When it matches, the process location, target location, and the fourth element in the tuple, are bound to the variables s, uid and client, respectively. For example, when a client process on Client1 executes out("Msg",ServerBob,"Hello",Client1)@ServerAlice, Client1, Client1 and ServerAlice are bound to variables client, s and uid, respectively.

Advice Lines 4-8 are the body of the advice that terminates the process if the target location of the out action (uid) is either ServerAlice or ServerBob, and the fourth element of the tuple (client) is not the location on which the process is running (i.e., s). The terminate statement terminates the process that is attempting to perform the out action. Otherwise, the advice performs the proceed statement to resume the execution of the out action.

Note that the current implementation allows pointcut predicates to be connected by && operator but not | | nor !. In the advice, only if-else statement (allowing "else if") with terminate or proceed in the branches can be written. It allows only one advice declaration per aspect. There is only one kind of advice [5].

3 Problems of Existing Approaches and Our Solutions

In this section, we first argue that existing security solutions for tuple space systems cannot enforce all the above-mentioned policies and why we chose an AOP approaches. Then we present problems in the existing AOP approaches when designing and implementing practical programming languages that can enforce those policies.

3.1 Associating Static Analysis and AOP

Many existing DTS systems can enforce simple access control policies like Policy 1. Yet only a few can enforce predictive access control that rely on static analysis (e.g., [13, 14]). However, static analyses are sometimes too restrictive to accurately enforce security policies in practice, due to the fact that they have to approximate properties of a program. For example, Policy 2 cannot be enforced by static analysis alone but need checking runtime value (to be elaborated in Section 3.3), thus existing approaches are incapable of enforcing them. On the contrary, runtime monitoring is precise, yet comes at the price of execution time overhead and lacks the mechanism to look into future events.

Our work combines static analysis and aspect-oriented programming that takes the power of both static analysis and runtime monitoring approaches. Additionally, AOP can help users separate security concerns.

3.2 Predicting Control- and Data-flows

Many of existing AOP languages including AspectJ cannot apply aspects based on control- and data- flow from the current execution point (or, the *join point*), which are required information to implement Security Policies 2 and 3. Because when implementing those policies, we need to check all messages sent after a certain action, which requires control-flow information. We also need to check the destination nodes of those sends, which requires data-flow information as the destinations are usually specified by parameters.

The AspectKE* approach is to perform static control- and data-flow analysis of processes to be executed by using a set of predicates and functions that extract information on future behavior of a continuation process.

[5] The full syntax of AspectKE* can be found in the other literature [34].

3.3 Combining Static and Dynamic Conditions

In order to implement some security policies, we need to check both static and dynamic conditions, which cannot be supported elegantly with existing approaches. For example, consider conformity of the following code fragment, which is modified from Listing 2 with Policy 2. Note that the value of userserver is given before execution.

```
1: in("Msg",userserver,friendserver,text)@self; (Step 8)
2: u=userserver;
3: out("Msg",friendserver,text,self)@u; (Step 9)
4: out(userserver,friendserver)@ServerAlice; (Step 9')
```

In order to judge conformity, we need to know, before executing Line 1, the destinations of message sends at Lines 3 and 4 are the same as the value in usersever. This however requires both static and dynamic checking. For Line 3, we need to statically analyze the program to determine if userserver and u refer the same value. For Line 4, we need to check that the runtime value of userserver is indeed ServerAlice.

Even in the AOP languages that support static program analyses, the users have to write a static analysis and a dynamic condition separately. This will make aspect definitions redundant and difficult to maintain.

The AspectKE* approach is to provide a *dual (static-dynamic) value evaluation mechanism* that can compare both results of static analysis and runtime values by executing a single comparison expression. We explain the mechanism in Section 4.3.

4 AspectKE*: Advanced Features

In this section, we illustrate how we addressed the above problems in AspectKE* along with aspects that implement two predictive access control policies.

4.1 Program Analysis Predicates and Functions

We introduce language constructs called the *program analysis predicates and functions* that predict future behavior of a program, and therefore are useful for enforcing predictive access controls that refer future events of a program.

Table 1 summarizes the predicates and functions, which allow for checking different properties of the future behavior of a continuation process; i.e., the rest of the execution from the current join point, or a process to be evaluated locally or remotely. In the table, z is the continuation process of the captured action. acts is a collection of action names such as IN and OUT. v is a variable (it shall be declared in the pointcut). locs is a collection of locations. When computing a predicate/function on process z, the results are collected from process z and all processes spawned by z. In Section 5, we will explain the implementation of those predicates and functions by using static analysis.

Table 1: Program Analysis Predicates and Functions

Predicate & Function	Return Value
performed(z)	the set of potential actions that process z will perform.
assigned(z)	the set of potential values that process z will use.
targeted(acts,z)	the set of destination locations that the actions in set acts of process z will target to.
used(v,acts, locs,z)	true if all potential actions acts in process z that use variable v are targeted only to locations in locs.

4.2 Aspects Protecting Passwords (Policy 3) and Chat Information (Policy 2)

Listing 4 demonstrates a use of program analysis predicate used, in an aspect that enforces Policy 3. This Policy terminates a process if particular data, is potentially output to an untrusted place. The aspect matches an in action for a login request, and checks if the continuation process sends the password only to the user's server but not to other locations.

```
 1  aspect protect_password{
 2    advice: in("Login",unbound location uid,unbound string pw)
 3      &&on(bound location s)&&target(bound location client)
 4      &&continuation(process z){                  // capture a continuation process
 5       if (element_of(client ,{ Client1 , Client2})&&     // check whether the target location is one of the clients
 6         !used(pw,{OUT},{uid},z))                  // check if the password is sent to locations other than
 7         terminate;                               // the user's server node
 8      else
 9         proceed;
10    }
11  }
```

Listing 4: Aspect for Protecting Password (Policy 3)

The pointcut of this aspect uses the *unbound* modifier for some of its parameters. The unbound modifier means that the variables are not bound to any value before the action is performed.

When a client performs an in action with a "Login" tag, the pointcut in Listing 4 matches it and binds Client1 to both s and client. It also records that variables uid and pw in the aspect are connected to unbound variables userserver and password in the client process. The variables uid and pw in the aspect are considered to have potential values that will be stored to the variables userserver and password in future. The predicate continuation captures the rest of the process, which is bound to variable z.

The body of the advice checks if the targeted location of in action is one of the clients (Line 5), and if the password is sent to locations other than the user's server node in the continuation process (Line 6). Here, the used predicate checks, if all the out actions that use pw (password) in process z has uid (userserver) as the destination. If not, the aspect terminates the client process. Since Client1 will send the password to Eavesdropper (at Line 8), the aspect will terminate the process at the in action at Line 5.

Note that the predicate checks the condition when variables pw and uid are not yet bound. The predicate therefore evaluates the condition with respect to the potential values bound in future.

At implementation-level, those potential values in the continuation process are the program locations collected by interprocedural data-flow analysis. For example, we can detect that userserver, assigned by the in action (at Line 5 in Listing 1), will be used not only within the continuation process of the same process (Lines 6, 8, 10 of process clientlogin in Listing 1), but also will in the processes spawned by this process (e.g., Line 6, 7 and 9 of process clientsendmsg in Listing 2).

Listing 5 shows an aspect that enforces Policy 2 by exploiting another program analysis function. In the aspect, the pointcut at Line 2 captures the in action in clientsendmsg (Line 6 of Listing 2). When the pointcut matches, values ServerAlice, Client1 and Client1 are bound to variables uid, s and client respectively.

The conditions at Lines 5 and 6 check whether the action reads from a client node, and the continuation process only sends messages to the user's server node (uid), which

```
1   aspect protect_message{
2     advice: in("Msg",bound location uid,location, string)&&
3       on(bound location s)&&target(bound location client)
4       &&continuation(process z){                        //capture a continuation process
5        if (element_of(client,{Client1,Client2})&&       //check whether the target location is one of the clients
6          ! forall (x,targeted({OUT},z))<x==uid>)        //check if the continuation process only sends
7          terminate;                                     //messages to the user's server node
8       else
9          proceed;
10    }
11  }
```

Listing 5: Aspect for Protecting Chat Information (Policy 2)

is specified by the second element in the tuple. First, the function targeted({OUT},z) at Line 6 returns all the destinations of out actions in process z. In the example, the destinations are potential values of userserver and Eavesdropper. Then the expression forall(x,...)<x==uid> checks if all the destination locations are the user's server node (uid). We shall further explain how this expression is evaluated in the following section.

4.3 Combination of Static Analysis and Runtime Checking

The above expression demonstrates how we uniformly perform static and runtime checking. When the advice runs at an in action, some of future out actions already have concrete destinations while others do not. AspectKE* can handle both cases. The expression x==uid holds either when the destination x of a future out action is predicted to have the same value as the one that is captured as uid, or when a future out action has a constant target location, which happens to be the same one in uid. Therefore, when advice captures the following action:

```
in("Msg", userserver, friendserver, text)@self;
```

where the value of userserver is ServerAlice, the expression x==uid holds for the destination of the following future action:

```
out("Msg", friendserver, text, self)@u;
```

because x and uid capture variables that have data-flow between them.

The expression x==uid also holds for the future action:

```
out(userserver,friendserver)@ServerAlice;
```

because x's runtime value is ServerAlice.

The aspect in Listing 5 suggests to *proceed* at Line 6 in Listing 2 when Alice executes the modified client program, however, it *terminates* the in action when users other than Alice executes this client program.

Note that sometimes it shall be able to simplify a combination of program analysis functions and basic predicates by using the used program analysis predicate. For example, forall(x, targeted(acts,z))<element_of(x,locs)> equals used(*,acts,locs,z). Thus the forall expression at Line 6 shall also be expressed by used(*,{OUT},{uid},z). We chose the formal one in our example because it can better illustrate what checks are performed in a decomposed manner.

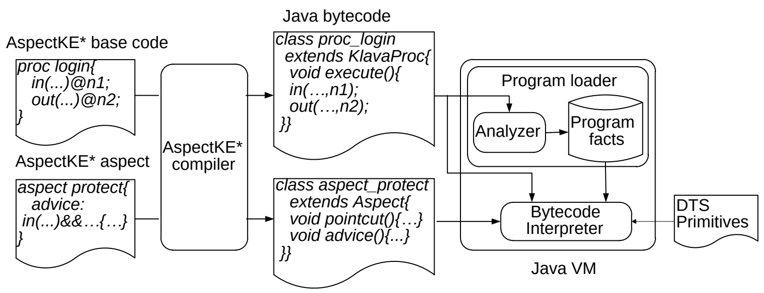

Fig. 2: Overview of the Implementation

5 Implementation

5.1 Overview

We implemented a prototype compiler and runtime system for AspectKE*, which are publicly available[6]. The compiler is written in 1618 lines of code on top of the ANTLR and StringTemplate frameworks. The runtime system is a Java package consisting of an analyzer and an bytecode interpreter. It is built on top of the Klava package [6] and ASM [8], with 6506 lines of Java code.

Figure 2 shows an overview of our implementation. The compiler generates a Java class for each node and process defined in the given base code. Aspects are translated into Java classes independently from the base code. The weaving process is carried out at runtime so that new aspects can be added to a running program without restarting. The analyzer implements a context-insensitive interprocedural data-flow analysis on Java bytecode. The results of the analysis, called *program facts*, are used for evaluating program analysis predicates and functions at runtime.

The architecture that analyzes Java bytecode at load-time fits the execution model of Klava which supports code mobility. In Klava, creation of a process at a remote node is realized by sending a Java class file to a Java virtual machine running at the remote node. Therefore, source code-level analysis and compile-time analysis are infeasible.

Compared to our previous naive implementation [34], the program facts avoid the overhead by not performing program analysis at runtime. When the runtime system loads the definition of a process, it analyzes the definition and extracts program facts for each action in the process. Later on, the advice body uses the program facts for evaluating program analysis predicates and functions. Note that our approach analyzes each process definition only once no matter how many aspects are applied to (any) actions in the process, and no matter how many program analysis predicates and functions are used and evaluated. In this way we minimize the overhead of the expensive program analysis. We confirmed this approach has better performance than the approach that analyzes program on-the-fly as AspectKE [34].

5.2 Dual-value Evaluation

Our language supports static and dynamic conditions in one expression by binding both static and runtime information to each variable in pointcut. Here we illustrate the underlying dual value evaluation mechanism by explaining how the condition at Line 6 in Listing 5 is evaluated with respect to process clientsendmsg in Listing 2 (except for the last eval action).

[6] http://www.graco.c.u-tokyo.ac.jp/ppp/projects/aspectklava.en

Labeling action parameters at compile-time. The compiler labels each parameter variable of any action in a process with a unique ID when translating the AspectKE* source code to Java bytecode. The labeled actions look like below. The labels will be used to represent program facts.

```
in("Msg",userserver¹,friendserver²,text³)@self⁴;
out("Msg",friendserver⁵,text⁶,self⁷)@userserver⁸;
out(userserver⁹,friendserver¹⁰,text¹¹)@Eavesdropper;
```

Extracting the program facts at load-time. When a node loads a process at runtime, the analyzer extracts the program facts for each action in the process and those processes under its control flow. A program fact contains primitive information about the program such as predicated dataflow *pdflow* and destination locations *dloc*.

For example, *pdflow* for the userserver at in action, namely $pdflow_{\mathsf{in}}$ contains {1,8,9} because userserver is used as the destination of the first out action and the first parameter of the second out action. *pdflow*s for other parameters and those in the two out actions are created similarly.

The *dloc*s of actions in the remaining process are computed with the help of *pdflow*. The analyzer first collects the set of labels and constants used as the destinations of actions, and then replaces each label in the set with the first label in the *pdflow* that contains it. Thus the destination location for the in action, namely $dloc_{\mathsf{in}}$, becomes {(OUT,1), (OUT,Eavesdropper)}, since the label for the first out action's destination is 8, which belongs to $pdflow_{\mathsf{in}}$ whose first element is 1.

Runtime pointcut matching and equality evaluation. When a node executes the in action at Line 6 in Listing 2, the pointcut in aspect protect_message in Listing 5 matches, and the condition forall(x,targeted({OUT},z)) <x==uid> is checked. Here, uid binds two values: one is a concrete value either ServerAlice or ServerBob, and the other is the label of the second parameter of this in join point action, i.e., 1. z binds the continuation process which yields, for targeted({OUT},z), {1, Eavesdropper} by simply referencing $dloc_{\mathsf{in}}$.

The interpreter checks for each element x in {1,Eavesdropper} if x is equal to uid, by comparing the uid's value (ServerAlice or ServerBob) and the label (1). When x is label 1, the equality holds. When x is Eavesdropper, the equality does not hold as it is compared against a runtime value.

6 Case Study on an EHR Workflow System

To assess applicability of AspectKE* to real world security policies, we implemented security policies for an *electronic healthcare record* (EHR) workflow system [34, 35] in AspectKE*.

The target system manages a database that stores patients' EHR records, where doctors, nurses, managers, and researchers need to rely them for performing different tasks. The target system and most policies are extracted from a health information system for an aged care facility in New South Wales, Australia [17]. We also incorporate security policies from the other literatures [9, 16], so as to examine basic access control and predictive access control policies.

The implemented EHR workflow system in AspectKE* consists of 16 nodes, 41 processes, and 23 aspects, totaling to 754 lines of code (496 lines for the target system and 258 lines for aspects).

Table 2: Natures and Implementation Status of Security Policies for EHR

#	operations	targets	judging properties	#aspects	LoC	program analysis
1	read/write/delete	EHRDB	doctor/nurse role	5	47	—
2	create/delete	RoleDB	manager role	3	33	—
3	read	EHRDB	attribute (doctor/nurse role)	5	51	—
4	read	EHRDB	location (nurse role)	2	41	—
5	remote evaluation	UserLoc	actions in migrating process	4	44	performed, targeted
6	read	EHRDB	actions in continuation process	4	42	used
7	read	EHRDB UserLoc	actions in continuation process actions in migrating process	—	—	used, targeted assigned

Table 2 summarizes the 7 security policies to be enforced to the target system with their implementation status. Column 1 denotes the policy number. Column 2-4 describes the nature (operations, targets and properties) of the policy. Columns 5 and 6 show the numbers and total lines of aspects for implementing the policy. The last column indicates the program analysis predicates or functions used in the aspects.

Policies 1-4 are basic access control, which regulates the rights of people with different roles to access patient's EHR records. We implemented them as 15 security aspects without using program analysis predicates and functions.

Policy 5 requires to handle process mobility. Among the 4 aspects, 2 aspects (for the eval action) use the program analysis predicates and functions in order to prevent potentially malicious process migration before its execution. Policies 6 and 7 are policies regarding the emerging use of data scenario. Before fetching an EHR record, it checks whether the continuation processes contain actions that illegally leak sensitive data of patients. For example, a researcher shall not leak patient names (part of an EHR record) to the public when doing his research. Policy 6 is implemented with 4 aspects by the program analysis predicates and functions.

We have not yet implemented Policy 7 because the current implementation of AspectKE* lacks a program analysis function assigned. We plan to provide this function in the future.

When using other AOP languages that support no analysis-based pointcuts (e.g., AspectJ), policies that depend on the classical access control models (Policies 1-4) can still be implemented, however policies that refer to predictive access control (Policies 5-7) are difficult to be implemented because they rely on future behavior of an action. (AOP languages with analysis-based pointcuts are discussed in Section 7.) When using other security mechanisms for tuple space systems, such as the ones based on Java Security framework [18] or other techniques [20, 21, 32], we could implement Policies 1-4. However, Policies 5-7 cannot be implemented because those mechanisms do not provide information on future behavior.

In summary, our experience shows that AspectKE* is expressive and useful to enforce complex real world security policies to a distributed system.

7 Related Work

Most existing AOP languages can only use merely past and current information available at the join point, but not future behavior of a program, in order to trigger execution of aspects. For example, `cflow` [23], `dflow` [26], and `tracematch` [1] are AOP constructs in AspectJ like languages that trigger execution of aspects based on calling-context, data-flow, and execution history, respectively, in the *past execution*, similar to the security enforcement mechanisms based on program monitors [3]. Those constructs would be useful to implement some of the security policies like Policy 1 in Section 2.3, but not so for Policies 2 and 3. A few AOP languages propose mechanisms by which aspects can be triggered by control flow of a program in the future, e.g, `pcflow` [22] and `transcut` [30], however, to use them for enforcing Policies 2 and 3 is difficult, due to their incapability to expose data-flow information in the future.

Even though several AOP extensions [2, 11, 25] offer the means of predicting future behavior, it is not easy to describe security policies because the users have to deal with low-level information. For example, SCoPE [2] allows the users to define pointcuts by using a user-defined static program analysis that is implemented on top of bytecode manipulation libraries. The users still have to develop the analysis at low-level. These languages also do not provide a mechanism to combine runtime data and static information as we do. In fact, our attempt showed that SCoPE can only partially implement Policy 2 but with much more complicated definitions [34]. Our approach offers better abstraction than existing analysis-based AOP languages using high-level predicates and functions. In particular, policies that require both runtime and static information cannot be easily implemented by others.

Alpha [29] provides sophisticated constructs to enforce policies we are interested in, but it lacks realistic implementation. AspectKE* can be considered as an approach to provide highly expressive pointcuts to AOP languages, such as `maybeShared` [7] `pcflow` [22], and the ones for distributed computing [27,28,31]. However, none are directly comparable to ours with respect to enforcement of security policies to distributed applications.

Many studies apply AOP languages to enforce access control policies [10, 15, 33]. To the best of our knowledge, only our approach supports predictive access control policies.

There are tuple space systems that provide security mechanisms. For example, SecOS [32] provides a low-level security mechanism that protects every tuple field with a lock. Secure Lime [21] provides a password-based access control mechanism for building secure tuple spaces in ad hoc settings. CryptoKlava is an extension to Klava with cryptographic primitives [4]. JavaSpaces [18], which is used in industrial contexts, has a security mechanism based on the Java security framework. Our work is different in using AOP with program analysis. Hence it not only provides a flexible way to enforce security policies, but also enables predictive access control policies, which cannot be realized in these approaches.

Some authors use static analysis on KLAIM based languages [13, 14]. They can be used to enforce very advanced security policies including a large set of predictive access control, however, they can not enforce policies (e.g., Policy 2) which requires accessing both static and runtime information. Additionally, users still have to explicitly annotate policies in the main code which our approach can avoid doing so.

8 Conclusions

We designed and implemented AspectKE*, which can enforce predictive access control policies to distributed applications. Our contributions can be summarized as follows. (1) Our approach can enforce predictive access control policies, which are difficult to be enforced in existing approaches. (2) We provide high-level program analysis predicates and functions that allow users to directly specify security policies in a concise manner. (3) The dual value evaluation mechanism enables to express a security condition that is checked either statically or dynamically by one expression. (4) We proposed an implementation strategy that combines load-time static analysis and runtime checking, which avoids analyzing programs at runtime. Further details can be found in the first author's dissertation [34].

Current AspectKE* language can merely make monitored processes terminate or proceed. We plan to extend the language so that it can perform other kind of actions. To do so, we need to incorporate effect from aspects while analyzing processes. The static analysis algorithm employed in current AspectKE* can only deal with explicit flows. Supporting indirect flows (e.g., dependency between processes that exchange information via tuples) is left for future work. To do so, we need to develop analysis techniques by combining pointer-analysis for tuple spaces with data- and control-flow analysis over processes.

Though AspectKE* is based on KLAIM, the techniques developed in this paper can also be applied to other distributed frameworks, especially those based on process algebra as well. We believe it is useful for monitoring, analyzing and controlling the behavior of mobile processes, under a distributed AOP execution environment.

Acknowledgements. This work is partly supported by the Danish Strategic Research Council (project 2106-06-0028) "Aspects of Security for Citizen". We would like to thank Lorenzo Bettini for discussing about the Klava system, and the members of the PPP group at the University of Tokyo for their comments on the work.

References

1. C. Allan, P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*, page 364. ACM, 2005.
2. T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD'07*, pages 161–172. ACM, 2007.
3. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *PLDI'05*, pages 305–314. ACM, 2005.
4. L. Bettini and R. De Nicola. A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces. In *FIDJI'02, Int. Workshop on scientific engineering of distributed Java applications*, pages 175–184. Springer, 2003.
5. L. Bettini and R. De Nicola. Mobile Distributed Programming in X-Klaim. In *Formal Methods for Mobile Computing, Advanced Lectures*, pages 29–68. Springer, 2005.
6. L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java package for distributed and mobile applications. *Software-Practice and Experience*, 32(14):1365–1394, 2002.
7. E. Bodden and K. Havelund. Aspect-oriented Race Detection in Java. *IEEE Transactions on Software Engineering*, 2010.
8. E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journees Composants 2002: Adaptable and extensible component systems*, 2002.
9. Canadian Institutes of Health Research. *Secondary Use of Personal Information in Health Research: Case Studies*. Public Works and Government Services Canada, 2002.
10. B. Cannon and E. Wohlstadter. Enforcing security for desktop clients using authority aspects. In *AOSD'09*, pages 255–266. ACM, 2009.
11. S. Chiba and K. Nakagawa. Josh: an open AspectJ-like language. In *AOSD'04*, pages 102–111. ACM, 2004.
12. R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

13. R. De Nicola, G. L. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.
14. R. De Nicola, D. Gorla, R. R. Hansen, F. Nielson, H. R. Nielson, C. W. Probst, and R. Pugliese. From flow logic to static type systems for coordination languages. In *COORDINATION'08*, pages 100–116. Springer, 2008.
15. A. S. de Oliveira, E. K. Wang, C. Kirchner, and H. Kirchner. Weaving rewrite-based access control policies. In *FMSE'07*, pages 71–80. ACM, 2007.
16. Department of Health, UK. *NHS Code of Practice-Confidentiality*, 2003.
17. M. Evered and S. Bögeholz. A case study in access control requirements for a health information system. In *ACSW Frontiers'04*, pages 53–61. Australian Computer Society, Inc., 2004.
18. E. Freeman, K. Arnold, and S. Hupfer. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, 1999.
19. D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
20. R. Gorrieri, R. Lucchi, and G. Zavattaro. Supporting secure coordination in SecSpaces. *Fundamenta Informaticae*, 73(4):479–506, 2006.
21. R. Handorean and G. Roman. Secure sharing of tuple spaces in ad hoc settings. *ENTCS*, 85(3):122–141, 2003.
22. G. Kiczales. The fun has just begun. *Keynote AOSD*, 2003.
23. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP'01*, pages 327–353. Springer, 2001.
24. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, pages 220–242. Springer, 1997.
25. G. Kniesel, T. Rho, and S. Hanenberg. Evolvable pattern implementations need generic aspects. In *RAM-SE'04*, pages 111–126. Universität Magdeburg, 2004.
26. H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *APLAS'03*, pages 105–121. Springer, 2003.
27. L. D. B. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *AOSD'06*, pages 51–62. ACM, 2006.
28. M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed AOP. In *AOSD'04*, pages 7–15. ACM, 2004.
29. K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP'05*, pages 214–240. Springer, 2005.
30. H. Sadat-Mohtasham and H. Hoover. Transactional pointcuts: designation reification and advice of interrelated join points. In *GPCE'09*, pages 35–44. ACM, 2009.
31. É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In *GPCE'05*, volume 3676 of *LNCS*, pages 173–188. Springer, 2005.
32. J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Science of Computer Programming*, 46(1-2):163–193, 2003.
33. B. D. Win, W. Joosen, and F. Piessens. Developing secure applications through aspect-oriented programming. In *Aspect-Oriented Software Development*, pages 633–650. Addison-Wesley, 2002.
34. F. Yang. Aspects with program analysis for security policies. Phd Dissertation, Technical University of Denmark, 2010.
35. F. Yang, C. Hankin, F. Nielson, and H. R. Nielson. Aspect-oriented access control of tuple spaces. Submitted to a journal.