

ConSpec – A formal language for policy specification[☆]

Irem Aktug^{a,*}, Katsiaryna Naliuka^b

^a Royal Institute of Technology, Stockholm, Sweden

^b University of Trento, Trento, Italy

ARTICLE INFO

Article history:

Received 1 April 2008

Accepted 3 September 2008

Available online 23 September 2008

Keywords:

Policy language

Security enforcement mechanisms

Runtime monitoring

Security frameworks

ABSTRACT

The paper presents ConSpec, an automata-based policy specification language. The language trades off clean semantics to language expressiveness; a formal semantics for the language is provided as security automata. ConSpec specifications can be used at different stages of the application lifecycle, rendering possible the formalization of various policy enforcement techniques.

© 2008 Elsevier B.V. All rights reserved.

1. Introduction

As mobile devices become increasingly popular, the problem of secure mobile application development gains importance. Mobile devices contain personal information, which users desire to protect. They also provide access to costly functionality, such as GSM services and GPRS connections. Hiding these resources from third-party applications would largely handicap application development for mobile platforms. It seems necessary to provide *controlled* access to the sensitive resources through fine-grained, at times application specific, constraints on execution.

A *security policy* selects a set of acceptable executions from all possible executions and is used to define how and under what conditions a sensitive resource can be accessed. For instance, a user policy may limit the number of SMSs that are sent by an application per hour in order to prevent spamming. The decision for allowing access to a requested resource at a certain point of the program execution may depend on various factors, such as the previous actions of the application, the state of the environment, the parameters of the request, etc. The user may want to forbid the sending of SMSs, for instance, after an application has accessed certain local files.

A program adheres to a policy if all its executions are in the set of executions selected by the policy. Several techniques exist to ensure that an application complies to a policy. *Static verification* techniques, such as model checking, analyze the program code in order to construct a mathematical proof that no execution of the program can violate the policy. Such an analysis is thorough and provides full assurance, at the same time, it is costly and often requires human interaction. *Runtime monitoring* can be used as an alternative to static checking. This security enforcement mechanism observes the behavior of a target program and terminates it if it does not respect the policy. Monitoring can effectively enforce many interesting security properties [16]. However, it creates performance overhead since each *security relevant action* of the program should be detected and checked against the policy. Monitoring may be performed *explicitly*, i.e. by a separate program which is *co-executed* (executed in parallel) with the untrusted application. Due to expensive interprocess communication however,

[☆] Partially supported by S3MS project (<http://s3ms.org>).

* Corresponding address: Royal Institute of Technology, Osquars Backe 2, 100 44 Stockholm, Sweden. Tel.: +46 704381234.

E-mail address: irem@nada.kth.se (I. Aktug).

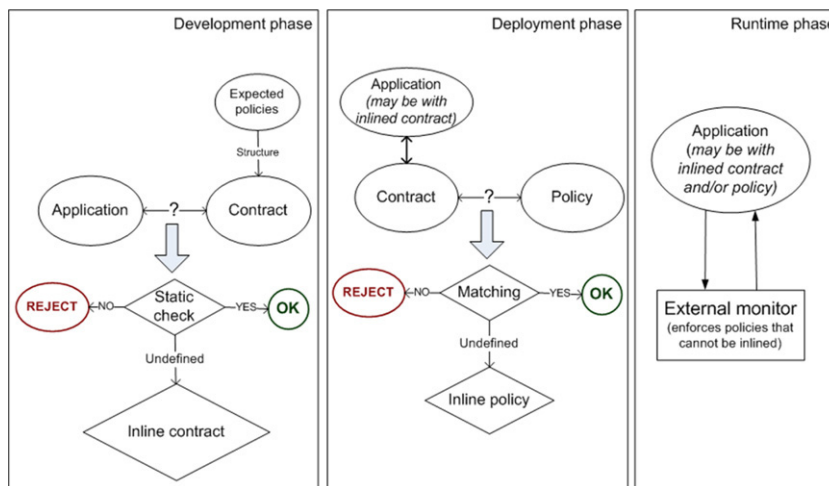


Fig. 1. Security enforcement through application development phases.

this technique is costly. In order to reduce this overhead, the monitor can instead be *inlined* in the untrusted program (see for instance [6]). Then, the code of the program is interleaved with the code of the monitor.

We describe here first how security specifications can be enforced at the three stages of the application lifecycle: the *development*, *installation* and *runtime* phases. The approach that we present here combines static verification and monitoring to enforce security properties on mobile devices in the most effective way. We associate with the application a *contract* [5], a piece of data that describes its *security-relevant behavior* and simplifies tasks related to security enforcement. A framework which spans different stages of the application lifecycle and combines different techniques for ensuring compliance benefits from a common language for policy specification. In turn, the different aspects of the framework imposes different restrictions on such a language. The main contribution of this paper is the language *ConSpec* (Contract Specification Language) which can be used for specifying both user policies and application contracts. A semantics for a subset of ConSpec is provided and the formal treatment of several activities in the framework is briefly explained based on this semantics.

The paper is structured as follows. In Section 2, we describe the lifecycle of the application paired with its contract. In Section 3, we discuss design decisions behind ConSpec, present its syntax and give a formal semantics to ConSpec. Discussion of the related work and final remarks Section 5 end the paper.

2. Security enforcement in the application lifecycle

In this section, we describe how security enforcement techniques can be applied throughout the lifecycle of an application and how the goals of all participants can be achieved in the contract-aware framework. The lifecycle of the application and the activities associated with each development phase are illustrated in Fig. 1. We make use of the following scenario in the rest of the section:

*Companies Alpha and Beta produce applications for mobile devices. Alpha develops application *Weather* that every morning at a user-defined time sends an SMS message to the operator's weather service and displays to the user the forecast it receives. Application *HappyBirthday*, produced by Beta, checks the user's address book and sends a congratulation SMS to each contact that has a birthday. Both companies want their applications to be used by as many users as possible.*

Alice is a mobile device user. She wants to download and use third-party applications. But she does not want these applications to break the following policy: "An application must not send more than 5 SMS messages per day". She also dislikes messages being sent to multiple addresses, as this can be used for spamming. So her other requirement is that "Each message can be sent to only a single phone number".

Development phase. We assume that the developer is aware of typical security policies and is willing to keep his application in conformance with them. It is sufficient that the developer is aware of which resources (and therefore which actions of the application) are security-relevant. Using this information he/she provides the application with a contract, which specifies the intended security-relevant behavior of the application. At this phase, the policy language is used for expressing this contract. The compliance of the contract and the application can be checked, for instance, using static verification by a trusted third party, who then signs the application and the contract by its private key. This analysis is performed by powerful machines rather than the mobile devices, and can make use of knowledge available to the developer (e.g. program specifications, annotations derived from the source code, etc.). Instead of code-signing, *proof-carrying code* (PCC) can be used to convey assurance in program-contract compliance [15]. The application and the contract are supplied with an easy-to-check proof of their compliance. Instead of statically verifying contract compliance, the compliance can be enforced at runtime by inlining a monitor for the contract in the application.

In our example scenario Alpha and Beta are not aware about the particular limit of SMS messages that Alice allows. But they know that the number of messages matters. Therefore, Alpha supplies the *Weather* application with the contract that the application sends only one message per day. Also, as the application sends messages only to the operator's weather service, they guarantee in the contract that all messages are sent to a single phone number. Whereas, Beta developers know that the messages that their application sends can be directed to various addresses. But, as the *HappyBirthday* application sends congratulations to each contact individually, the contract states that each message is directed to one phone number. However, Beta developers still cannot tell in advance how many messages their application sends per day. For this reason, the contract for their application is more complex. It says that the application will send one message to every contact from the address book that has a birthday.

Installation phase. Before the program is installed on the device, a formal check is needed to show that the security-relevant behavior of the application given by the contract is acceptable by the user policy. If policies and contracts are captured with automata on infinite strings, the problem of matching a policy against a contract reduces to the language containment problem for such automata. The complexity of this task severely restricts the expressive power of the policy language (for example, the problem of language containment is undecidable for two context-free languages [10,11]). When the problem is decidable, however, contract-policy matching is much simpler than checking the program against the policy itself, and is more likely to be feasible on a mobile device [5].

In our scenario, the contract of *Weather* can be matched against Alice's policy "No more than 5 messages per day". Also the rule "All messages are sent to a single number" matches Alice's requirement "Each message is sent to a single number". So this application is permitted to run at the device without any modifications. *HappyBirthday*'s contract also satisfies this second requirement. Its contract can still not be matched against the entire policy due to the lack of information on the precise number of messages the program can send. It can run on the device only if it undergoes inlining.

A policy that is not covered by the contract can be enforced by monitoring. If the program is to be monitored explicitly, hooks that notify the monitor about ongoing security-relevant activity should be injected to the program in the installation stage, i.e. prior to execution. If the monitoring task is to be optimized in order to reduce runtime overhead, the monitor for the desired policy should be inlined into the program at this stage.

For instance, application *Weather* does not need to undergo inlining since its compliance to the user's policy has already been verified. But a monitor for Alice's policy can be inlined in the *HappyBirthday* application to ensure that the application does not send more than 5 messages.

Runtime. At runtime, a security policy can be enforced on the application by monitoring. Because of the performance overhead created by monitoring, it is preferable to use static methods described above and leave as little work to runtime as possible. But in many cases, the application of other techniques is not feasible (or not even possible due to, for example, the unavailability of the source code), and runtime monitoring is the only solution to protect a system.

In our setting, application *Weather* will not be monitored. But *HappyBirthday* will, and if a violation is detected (that is, if the application is trying to send the 6th message), it will be terminated. The behavior of the program will otherwise be unaltered (except for the slight performance deterioration due to monitoring) and the user will be able to freely enjoy its functionality.

3. ConSpec language

ConSpec is a policy language intended for programs written in intermediate object-oriented languages such as the bytecode languages of Java and .NET. Security relevant actions are taken as method invocations, more specifically system calls or invocations of API methods. The intention behind ConSpec is to design a language that can be exploited both for specification of requirements and for the description of the security-relevant behavior of actual systems. For this reason, the formalism selected is based on automata, which have been used for both purposes. For instance, the SPIN tool [9] inputs system specifications as models written in the guarded-command language Promela and performs model checking on the Büchi automata extracted from these models. Security properties are also expressed as automata in various approaches (e.g. [16,17]).

ConSpec is strongly inspired by the policy specification language PSLang, which was developed by Erlingsson and Schneider [6] for runtime monitoring. PSLang policies consist of a set of variable declarations, followed by a list of security relevant events, where each event is accompanied by a piece of Java-like code that specifies how the security state variables should be updated in case the event is encountered in the current state. PSLang policies make monitor inlining simple: the updates provided by the user can be almost directly inserted into the target program. However, this leads for making specifications less formal. A policy text is intended to encode a security automaton: the state variables represent the automaton states and updates represent transitions. While this intuition is given, the exact way to extract the automaton from a PSLang policy is not provided. Such a task is not trivial due to the power of the programming language constructs that can be used in the updates.

Further we provide a formal semantics which maps ConSpec policies, which are on single executions of an application, to formal objects that can be used in constructing mathematical proofs. It is important to note that ConSpec is a more restricted language than PSLang; this is a design decision taken in order to allow application of formal methods for all stages of the development process, and not just runtime monitoring. More specifically, ConSpec does not allow arbitrary types in representing the security state and restricts the way the security state variables are updated. We have used a guarded-command language for the updates where the guards are side-effect free and commands do not contain loops. The simplicity of the language then allows for a comparatively simple semantics. While the general ConSpec, which is the common language

for all tasks in the application lifecycle, is to be kept as simple as possible, specific tasks may allow certain extensions. For instance, while putting conditions on heap objects make matching undecidable, these are easily handled by monitor inlining. A table that shows which features can be supported by different tasks is included in the [Appendix](#).

Example 1. Assume method `Open` of class `File` is used for creating files (when argument `mode` has value “CreateNew”) or for opening files (`mode` is “Open”), either for reading (argument `access` is “OpenRead”) or for writing.¹ Assume further that method `Open` of class `Connection` is used for opening connections, that method `AskConnect` is used for asking the user for permission to open a connection and that this latter method returns true in case of approval. Now, consider the security policy, which allows applications to access existing files for reading only, and requires, once such a file has been accessed, applications to obtain approval from the user each time a connection is to be opened. The policy also does not allow the application to execute further if a file opening operation raises an exception. This policy can be specified in `ConSpec` as follows:

```
SCOPE Session

SECURITY STATE
    bool accessed= false;
    bool permission = false;

BEFORE File.Open(string path, string mode, string access)
PERFORM
    mode.equals("CreateNew")           -> { skip; }
    mode.equals("Open") && access.equals("OpenRead") -> { accessed= true; }

BEFORE Connection.Open(string type, string address)
PERFORM
    !accessed           -> { permission = false; }
    accessed && permission -> { permission = false; }

AFTER bool answer= GUI.AskConnect() PERFORM
    answer -> { permission=true; }
    !answer -> { permission=false; }
```

We begin by specifying that the policy applies to each single execution of an application. Scope declaration is followed by the *security state declaration*: the security state of the example policy is represented by the boolean variables `accessed` and `permission`, which are both false initially to mark, respectively, that no file has been accessed and that no permissions are granted when the program begins executing. The example policy contains three *event clauses* that state the conditions for and effect of the security relevant actions: call to the method `File.Open`, call to the method `Connection.Open` and return from the method `GUI.AskConnect`. The types of the method arguments are specified along with representative names, which have the event clause as their scope. The *modifiers* `BEFORE` and `AFTER` mark whether the call of or the normal return from the method specified in the event clause is security relevant (exceptional returns can be specified by the modifier `EXCEPTIONAL`). Event clauses contain guards and associated updates to the security state variables.

The restriction that the application should not overwrite local files is specified by the first event clause. Whatever the execution history, whenever the application calls the method `File.Open`, it should be creating a new file (the first guard) or it should be opening an existing file for reading (the second guard). If neither of these conditions hold for the current call. In order to decide if the application is allowed to open a connection, history of existing file accesses and user permissions is consulted. An access to an existing file through a call to `File.Open` is recorded by the update block of the second guard in the event clause. If the current history contains an access to an existing file (that is if `accessed` is true), then an attempt to open the connection is allowed only if the security state variable `permission` is true. The only way this variable is true at a certain point in the execution is, if the last execution of the method `GUI.AskConnect` has returned “true”. Notice that the policy does not allow the same permit to be used for several connections since `permission` is set to false before each connection.

3.1. ConSpec syntax

[Fig. 2](#) summarizes the syntax of `ConSpec` policies.

States. The security state variables of `ConSpec` are restricted to the primitive types (*PrimType*): booleans, integers, and strings. `ConSpec` files set a limit on values of the type `int` which consist of some initial segment of natural numbers. Similarly, maximum length of strings are specified. (We have skipped these in the example policy.) This aims to limit the state space of

¹ The methods used in this policy are not part of any standard API but have been chosen for the sake of the example.

```

MAXINT M
MAXLEN N
SCOPE <Object ClassName PersistentStateDec
  | Session
  | Multisession PersistentStateDec
  | Global PersistentStateDec>

SECURITY STATE
  PrimType SecVar1 = InitVal1
  ⋮
  PrimType SecVarN = InitValN

Clause1
  ⋮
ClauseK

```

(a) Policy syntax.

```

<BEFORE
|EXCEPTIONAL
|AFTER [Type Name = ]> Signature

PERFORM
  Guard1 -> {UpdateBlock1}
  ⋮
  GuardM -> {UpdateBlockM}
  [ ELSE -> {UpdateBlock} ]

```

(b) Event clause syntax.

Fig. 2. ConSpec syntax.

the corresponding automata, in order to enable matching. The Scope construct is used for expressing security requirements on different levels and explained in more detail below.

Event clauses. An event clause (Fig. 2(b)) gives us a security relevant action and its modifier. Security relevant events are bound to the API methods in the program (we support programs delivered in either Java or .NET intermediate bytecode language). In order to resolve which method is of interest in case of overloading, the argument types of the method is to be specified as part of the action specification. The security relevant action is then fully specified by its *signature* which consists of the name of the method, the class to which the method belongs and the types of its arguments. The signature of an event clause is defined as the signature of the method associated with it. In ConSpec policies, all event clauses with the same modifier have a unique signature. This restriction means that one can not, for example, have two BEFORE clauses for the same method and has been imposed in order to ensure determinism. Notice that since the signature does not include the type of the return variable, it is not possible to have two AFTER event clauses for the same method, even if they do not agree on the return variable types. The modifier states when the update to the state will be performed: before the event, after the event or immediately after the throwing of an exception by the event.

Guards and update blocks. The event specification is followed by a sequence of guard-update block pairs. The update specifies how a state will be updated for the security relevant action while the guard selects the states, which the particular update will apply, as a subset of all states. The guards are evaluated top to bottom and the update corresponding to the first guard that holds, is performed. In case none of the guards evaluates to true, there is no transition for that action from the current state, unless an ELSE block is present, in which case the update of this block is executed. The guard is a side-effect free boolean expression which can mention only the set of argument values (and the return value when the AFTER modifier is used), and the security state. The update block begins with declarations of the local variables, which have the current block as their scope. A list of assignments to local variables and security state variables follow the declarations. If no assignments are present, the update block consists of the statement `skip`. The expression language used for forming guards and right hand side of assignments are explained below.

Expressions. The expression language of ConSpec has been designed to ensure that checking language containment of the induced automata (the matching problem) is decidable. Variables except security state and local variables can be of any type in *Type*, which includes both primitive types and classes. The expressions on integers are built using basic arithmetic and comparison operators. Strings can be checked for equality and the prefix relation using the functions `equals` and `beginsWith`, respectively. The expression language of ConSpec can potentially be extended with calls to other side-effect free functions. Expressions can also include field accesses using object references, expressed by the “.” operator. Regardless of the modifier used, accesses to fields of method arguments are interpreted on the heap at the time of call, while accesses to fields of the return value are interpreted at the time of return. Therefore, it is not, as of yet, possible to put constraints on the fields of a method argument at the time of return. The last field accessed in a field access expression should always be of one of the primitive types.

Example 2. Consider the contract of the Weather application described in Section 2. The contract imposes two restrictions on the program execution: it states that only one message can be sent by the application each day, and that all messages are directed to a single phone number, that cannot be changed during the program lifetime. We assume that this is the first number the message is sent to. The corresponding contract is expressed in ConSpec as shown below.

The security state of this contract includes the integer variable `lastmessageday` that stores the date the last message was sent, and the string `usednumber`, which is used to record the phone number of the first recipient.

The example contract contains one event clause bound to the .NET API call `WindowsMobile.PocketOutlook.SmsMessage.Send()`. This API call does not contain parameters. Particulars of, and all particularities about the message are stored in the fields of the `SmsMessage` object itself. In the contract below, the first guard-update block pair relates to

the case when the application sends a text message for the first time. In this case the string `usednumber` is not associated with a value yet, so we check that the message is directed to one number only, and record this number and the current date for future comparison. If it is not the first time the message is being sent, we check that the date has changed since the time the last text message was sent in the second guard-update block pair. We also check that the message is directed to only one number, and that this number is the one that we recorded previously. If these conditions are met then the message is allowed to be sent, and the date is again recorded for future use. The contract is violated in an execution if and only if neither of the guards are satisfied by a call to the security-relevant method in the execution. This indicates that the application tried to send a message when it was forbidden to do so by the security requirements.

```
SCOPE Session
SECURITY STATE
    int lastmessageday = 1;
    string usednumber = " ";

BEFORE WindowsMobile.PocketOutlook.SmsMessage.Send()
PERFORM
    usednumber == " " &&
    this.To.Count == 1      -> { lastmessageday = Now.GetDay();
                               usednumber = this.To[0]; }

    this.To.Count == 1 &&
    this.To[0] == usednumber &&
    !Now.GetDay().equals(lastmessageday -> { lastmessageday = Now.GetDay(); }
```

In this example, we have used two extensions to ConSpec. One is the use of array variables. The other is the occurrence of (side-effect free) API methods in guards and update blocks. Both extensions are easily accommodated in ConSpec.

Scopes. Case studies show that many interesting real-life policies concern the entire execution history rather than a single run of the application [19]. However, most policy languages (including PSLang) do not contain the feature of distinguishing between events in the current run and in the previous runs. ConSpec is expressive enough to write policies on multiple executions of the same application (scope `Multisession`) and on executions of all applications of a system (scope `Global`), in addition to policies on a single execution of the application (scope `Session`) and on lifetimes of objects of a certain class (scope `Object`). The syntax of persistent state declaration is similar to security state declaration and aims to specify the state that is preserved across single executions when the scope is `Multisession` or `Global`. When the scope is `Object`, the security state declaration specifies the state local to each object of the class, while the persistent state is equivalent to the security state of scope `Session`, that is the security state of the application for a single execution.

Example 3. The personal information manager (PIM) saves personal information (e.g. phonebook) in mobile devices. Secure connections are established by connecting to destinations starting with “https://”. The policy “An application must not access the PIM while unsecure connections are open and must only open secure connections after the PIM is accessed” is expressed by the following ConSpec policy:

```
SCOPE Object Connection

PERSISTENT SECURITY STATE
    bool opened = false;

SECURITY STATE
    bool secure = false;
    bool active = false;

BEFORE PIM.open()
PERFORM
    secure || !active -> { opened = true; }

BEFORE Connection.open(string url)
PERFORM
    !opened && url.startsWith("https") -> { active = true; secure = true; }
    !opened && !url.startsWith("https") -> { active = true; secure = false; }
    opened && url.startsWith("https") -> { active = true; }

AFTER Connection.close()
PERFORM
    true -> { active = false;}
```


- An action $a = \langle \mathbf{c}, \mathbf{m}, (v_1, \dots, v_n), h \rangle$ is a security relevant pre-action, if and only if the ConSpec policy contains an event clause with the BEFORE modifier and the event $\mathbf{c.m} (\tau_1 x_1, \dots, \tau_n x_n)$,
- Similarly, an action $a = \langle \mathbf{r}, \mathbf{c}, \mathbf{m}, (v_1, \dots, v_n), h, h' \rangle$ where \mathbf{r} is a value v or ε is a security relevant (non-exceptional) post-action, if and only if the ConSpec policy contains an event clause with modifier AFTER followed by “ $\tau x =$ ” and the event $\mathbf{c.m} (\tau_1 x_1, \dots, \tau_n x_n)$ or only the event $\mathbf{c.m} (\tau_1 x_1, \dots, \tau_n x_n)$, respectively.
- Finally, an action $a = \langle \text{exc}, \mathbf{c}, \mathbf{m}, (v_1, \dots, v_n), h \rangle$ is a security relevant (non-exceptional) post-action, if and only if the ConSpec policy contains an event clause with modifier EXCEPTIONAL and the event $\mathbf{c.m} (\tau_1 x_1, \dots, \tau_n x_n)$.

In all three cases, the types of the arguments (and the type of the return value) specified in the event clause should match the components of the action, i.e. $v_1 \in \|\tau_1\|, \dots, v_n \in \|\tau_n\|, v \in \|\tau\|$.

Notice that the set of actions is to be a countable set, although it may be infinite (for instance if one of the arguments of a method is not of a primitive type). Care must be taken on the heap components when constructing the set of actions. In order to assure countability, the set of all heaps \mathbb{H} can be taken as the set of all partial heaps, where a partial heap is the part of the heap reachable through a field access. It is always possible to determine the field access expression with the most number of accesses in a policy and use this depth to bound the size of the set \mathbb{H} .

Transitions. Each event clause of the policy induces a partial transition function. The transition functions δ^b and δ^\sharp of the automaton are the union of the partial functions corresponding to event clauses with the BEFORE and AFTER/EXCEPTIONAL modifier, respectively. The definition of the partial functions is similar for both types of event clauses. For brevity, here we only describe the latter case, which is slightly more general.

Consider an AFTER event clause ϕ^\sharp :

```

AFTER   $\tau r = \mathbf{c.m} (\tau_1 x_1, \dots, \tau_n x_n)$ 
PERFORM
   $G_1 \quad \rightarrow \quad U_1$ 
       $\vdots$ 
   $G_m \quad \rightarrow \quad U_m$ 

```

Let $AVar = \{x_1, \dots, x_n\}$ be the set of formal arguments of the event and $PVar = \{r\} \cup AVar$ be the set of all program variables of the event clause. Below, let states $q \in Q$ be as defined above, and let $\sigma : PVar \rightarrow Val$ range over the set Σ of mappings from program variables to values which respect the declared types of the variables. For guards G_j and update blocks U_j of the event clause, we assume the semantic functions:

$$\begin{aligned} \llbracket G_j \rrbracket &: Q \times \Sigma \times \mathbb{H} \times \mathbb{H} \rightarrow \{\text{true}, \text{false}\} \\ \llbracket U_j \rrbracket &: Q \times \Sigma \times \mathbb{H} \times \mathbb{H} \rightarrow Q \end{aligned}$$

where the two heaps in the function types refer to the heap of the program before and after the execution of the method call, respectively. The ELSE keyword used as a guard would then correspond to the guard true.

Semantics of field access expressions occurring in guards and update blocks are relativized on heaps. The heap is not changed by the automata, but is used to look up fields of object references. To fetch the field values of method arguments, the heap at the time of call is accessed. In order to get the field values of the return value, however, heap at the time of return is used. Below, we denote the heap before the call with h^b , and the heap after the call with h^\sharp . Then, the value of a field access expression with depth k is as follows:

$$x.\mathbf{f}_1.\mathbf{f}_2 \dots \mathbf{f}_k = \begin{cases} (h^b (\dots (h^b (h^b \sigma(x) \mathbf{f}_1) \mathbf{f}_2) \dots) \mathbf{f}_k) & \text{if } x \in AVar \\ (h^\sharp (\dots (h^\sharp (h^\sharp \sigma(x) \mathbf{f}_1) \mathbf{f}_2) \dots) \mathbf{f}_k) & \text{if } x = r. \end{cases}$$

Then, the event clause ϕ^\sharp induces a partial mapping $f_\phi^\sharp : Q \times A^\sharp \rightarrow Q$ as follows. For a security state q and postaction $a = \langle v, \mathbf{c}, \mathbf{m}, (v_1, \dots, v_n), h^b, h^\sharp \rangle$, we define $f_\phi^\sharp(q, a) = q'$ if there exists $1 \leq i \leq m$ such that:

- $\llbracket G_j \rrbracket(q, \sigma, h^b, h^\sharp)$,
- $\forall i < j. \neg(\llbracket G_i \rrbracket(q, \sigma, h^b, h^\sharp))$ and
- $\llbracket U_j \rrbracket(q, \sigma, h^b, h^\sharp) = q'$

where $\sigma : PVar \rightarrow Val$ is the valuation given by the correspondence of actual (v_1, \dots, v_n, v) to formal parameters (x_1, \dots, x_n, r) . This definition captures that the guards are evaluated in order from top to bottom in order to select the right update block.

Finally, the post-transition function δ^\sharp is the union of the functions induced by each event clause (with disjoint domains):

$$\delta^\sharp = \bigcup_{\phi^\sharp \in \mathcal{P}} f_{\phi^\sharp}.$$

4. ConSpec in use

The main advantage of ConSpec is that it allows for a formal treatment of the various enforcement techniques mentioned in Section 2 through its automata-based semantics. Here we briefly explain how this can be achieved.

Matching. One way to match a ConSpec contract against a ConSpec policy is to check that the language of the contract automaton is included in the language of the policy automaton. Since the domains of the security state variables are bounded, the extracted automata have finitely many states (but possibly infinitely many transitions) and standard methods for checking language inclusion for automata (see for instance [4]) can be facilitated for contract-policy matching. Such an approach is taken in [14] for matching contracts against policies, when both are expressed as *automata modula theory* (\mathcal{AMT}), a type of symbolic Büchi automata. ConSpec policies can be converted to \mathcal{AMT} 's in order to make use of the matching algorithms provided in [14].

Monitoring. Given a program and a ConSpec policy with scope `Session`, the concept of monitoring can be formalized by defining the *co-execution* of the corresponding ConSpec automaton with the program [1]. Such co-executions are a subset of the set of interleavings of the individual executions of the program and the automaton. Co-executions satisfy the following condition: when the execution of the program component is projected to its security relevant action executions, each pre-action is immediately preceded by a transition of the automaton for the same action; dually, each post-action is immediately followed by a corresponding automaton transition. Therefore, it is simple to show that the program component of the co-execution adheres to the given policy, as the co-execution includes an accepting trace of the automaton for the program execution.

Monitor inlining. Inlining a ConSpec policy with scope `Session` can be performed similar to inlining a PSLang policy (see [6] for details on inlining PSLang policies). A class definition is added to the target program which stores the security state variables. Then the program is rewritten so that each security relevant method call is wrapped with code compiled from the corresponding event clause(s) of the policy. Such a code segment evaluates the guards of the event clause from top to bottom and executes the updates associated with the first guard that is satisfied. If none of the guards evaluate to true, the program is terminated. The modifier of the event clause determines where this segment is placed relative to the method call.

In [1], the inlining is performed by inserting the check/update code in the methods of the untrusted program, around method invocation instructions that may call security relevant API methods. It is also possible to monitor the program by “compiling” the security policy into a single module which is included in the program. This module is enquired prior to all security relevant method calls. The module then keeps the security state and performs all checks and updates. The original program methods are then altered only by calls to interface methods of the module. This approach simplifies proving inlining correct as all security code resides in one module. Vanoverbergh and Piessens describe such an inliner in [18].

The correctness of various monitor inlining schemes can be proven by setting up a bisimulation relation between the states of the inlined program and the states of the co-execution of the original program with the ConSpec automaton (of the policy).

Verification of monitor inlining using PCC. Showing that the program has an inlined monitor for the contract it guarantees is much less an effort than showing the correctness of a particular inlining algorithm. In [1], an annotation scheme is described which characterizes, in terms of JVM class files annotated by formulas in a suitable Floyd-like program logic, policy-adherence of the program and the existence of a concrete representation of the monitor state inside the program. The verification of a concrete monitor inliner then reduces to proof of validity of the corresponding annotations. It is also sketched how the annotations can be completed automatically, for a simple inliner, to produce a fully annotated program. This process can then be used, provided a bytecode weakest precondition checker, in a proof-carrying code setting to certify monitor compliance to a third party such as a mobile device. The authors prove the correctness of their inlining algorithm using this annotation scheme, i.e. show that each program which has been inlined for policy \mathcal{P} validates the assertions produced using \mathcal{P} .

Static analysis. Verification of safety properties is a well-studied subject. The method that is most commonly used for this purpose is model checking, where a model of both the program and the property are constructed. Then various techniques are used to check whether the program is a model for the temporal logic formula that represents the property. For ConSpec properties, a suitable method is to use pushdown model checking. Alternatively, the annotation scheme described above can be used to statically verify contract adherence. The assertions produced characterize all programs which have a concrete representation of the monitor. When the program has been inlined, references to this concrete representation in the assertions are instantiated by the inlined variables. For an arbitrary, contract-adherent program, the verification step would involve finding suitable candidates for the concrete representation of the monitor as a function of the program state, besides proving the resulting verification conditions.

5. Related work

There exists a number of automata-based languages for security policy specification. Amongst these, ConSpec is closest to PSLang [6] which has also introduced the modifiers used in ConSpec. The language is intended solely for runtime monitoring and freely uses programming language constructs such as abstractions and functions. This enables a larger class of policies to be specified but also complicates the task of providing a formal semantics. Since the authors do not provide such a formalisation, their monitor inlining algorithm for PSLang is to be trusted on intuition as no proof of its correctness can be constructed.

The Polymer language [2] has the same drawback. Polymer policies consist of Java classes which, when inlined, may trigger various action in case of violation. For instance it is possible to execute some recovery action as a response to the violation, after which the application is allowed to progress. Polymer policies implement *edit automata* [13], which extend security automata [7]. But the correctness of the Polymer policy inlining cannot be proven either, as its semantics is not formally presented. In a recent work [3], a simpler version of the Polymer language is presented with its semantics, in the context of a lambda calculus. Using this semantics, the authors prove *uncircumventability* of the monitor, i.e. that the untrusted program can not perform security relevant actions by circumventing the monitor. However, the result applies to the toy language and not to any Polymer implementation.

Many logic-based formalisms are used to express security properties for monitoring purposes (e.g. [8]). Temporal logic formulae can be translated to automata by applying, for instance, a tableaux procedure [12]. It is therefore conceivable that temporal logic formulae being translated to ConSpec, so to enable the use of this type of formalism in the framework.

Model-carrying code (MCC) [17] method is based on the idea of supplying untrusted code with additional information to simplify its verification against user policies. In MCC, this additional information is an extended FSA (EFSA) that represents the model of the program. This approach has much in common with ours, and EFSA is very similar to ConSpec automata. The difference is that EFSA allows for unbounded variables to be used as the security state, but in turn allows only equality conditions for comparing variables. ConSpec allows, however, more sophisticated expressions, including basic arithmetic operations and comparisons of numeric values. Also, our framework does not rely entirely on monitoring for enforcing code-contract compliance. In many cases the compliance can be verified statically and run without performance overhead.

6. Conclusion

In this paper, we present the policy language ConSpec, which has been designed for formalizing security requirements as well as representing the security-relevant behavior of the application. ConSpec specifications can be used for various tasks during all stages of the application lifecycle to ensure that the application conforms to the user policy. The main features of ConSpec are this universality and its tight connection with the underlying formalism, which is a fundamental component of any formal proof of policy adherence.

The semantics presented in this paper covers only policies for a single execution of the untrusted program. We leave devising semantics for policies with other scopes (i.e. `MultiSession`, `Object` and `Global`) to future work. In the scope of the S3MS project, we have formalized techniques for enforcing ConSpec policies on sequential programs as summarized in Section 4. Efficient static verification of ConSpec properties is another interesting subject to explore. The main challenge, however, is extending the approach to handle multi-threaded applications. Such a setting brings about synchronization issues as mutually dependent events may occur in different threads and data used by the monitor for decision-making may be shared between threads.

Acknowledgements

The authors thank Dilian Gurov and Fabio Massacci for valuable comments and discussions.

Appendix. ConSpec features for different tasks

As extensions to the current ConSpec language, introducing object reference and list types as security state types emerge as beneficial features considering real-life policies. The list type makes simple iteration meaningful to include in the update language, to enable, for instance, updating all elements. The update language then can be extended with a simple construct that iterates over flat lists. Extensions to the language should be considered thoroughly, as these may introduce undecidability of various tasks identified in our framework. Below we provide a table that shows which extensions to the language can be handled by which tasks in the framework. The constructs of ConSpec are specified in the rows of the tables below, whereas the activities are specified in the columns.

Construct	Static analysis	Monitoring	Matching
Policy scope			
Scope object	+	+	+
Scope session	+	+	+
Scope multi session	-	+	+
Scope global	-	+	+
State declaration			
Bounded integers	-	+	+
Bounded strings	-	+	+
Booleans	+	+	+
Object references	-	+	-
Lists (of the above)	-	+	-
Unbounded versions of the above types	-	+	-
Command			
Assignment	+	+	+
Conditional branch	-	+	-
For-loop	-	+	-

References

- [1] I. Aktug, M. Dam, D. Gurov, Provably correct runtime monitoring, in: Proceedings of the 15th International Symposium on Formal Methods, FM'08, in: Lecture Notes in Computer Science, vol. 5014, Turku, Finland, May 2008, pp. 262–277.
- [2] L. Bauer, J. Ligatti, D. Walker, Composing security policies with Polymer, in: Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation, 2005, pp. 305–314.
- [3] L. Bauer, J. Ligatti, D. Walker, Composing expressive run-time security policies, ACM Transactions on Software Engineering and Methodology, 2008 (in press).
- [4] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal Methods in System Design 1 (2–3) (1992) 275–288.
- [5] N. Dragoni, F. Massacci, K. Naliuka, I. Siahaan, Security-by-contract: Toward a semantics for digital signatures on mobile code, in: European PKI Workshop: Theory and Practice, 2007 (in press).
- [6] Ú. Erlingsson, The inlined reference monitor approach to security policy enforcement, Ph.D. Thesis, Dep. of Computer Science, Cornell University, 2004.
- [7] K.W. Hamlen, G. Morrisett, F.B. Schneider, Computability classes for enforcement mechanisms, ACM Transactions on Programming Languages and Systems 28 (1) (2006) 175–205.
- [8] K. Havelund, G. Rosu, Efficient monitoring of safety properties, International Journal on Software Tools for Technology Transfer (STTT) (6–2) (2004) 158–173.
- [9] G.J. Holzmann, The model checker SPIN, Software Engineering 23 (5) (1997) 279–295.
- [10] J.E. Hopcroft, On the equivalence and containment problems for context-free languages, Theory of Computing Systems 3 (2) (1969) 119–124.
- [11] H.B. Hunt, D.J. Rosenkrantz, On equivalence and containment problems for formal languages, Journal of ACM 24 (3) (1977) 387–396.
- [12] Y. Kesten, Z. Manna, H. McGuire, A. Pnueli, A decision algorithm for full propositional temporal logic, in: CAV, 1993, pp. 97–109.
- [13] J. Ligatti, L. Bauer, D. Walker, Edit automata: Enforcement mechanisms for run-time security policies, International Journal of Information Security 4 (1–2) (2005) 2–16 (Published online 26 Oct 2004).
- [14] F. Massacci, I. Siahaan, Matching midlet's security claims with a platform security policy using automata modulo theory, in: in Proc. of The 12th Nordic Workshop on Secure IT Systems, NordSec'07, October 2007.
- [15] G.C. Necula, Proof-carrying code, in: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, New York, NY, USA, 1997, pp. 106–119.
- [16] F.B. Schneider, Enforceable security policies, ACM Transactions on Infinite Systems Security 3 (1) (2000) 30–50.
- [17] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, D.C. DuVarney, Model-carrying code: A practical approach for safe execution of untrusted applications, ACM SIGOPS Operating Systems Review (2003).
- [18] D. Vanoverberghe, F. Piessens, A caller-side inline reference monitor for an object-oriented intermediate language, in: Proc. of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems, FMOODS'08, Oslo, Norway, June 4–6, 2008, in: Lecture Notes in Computer Science, vol. 5051, Springer, 2008, pp. 240–258.
- [19] A. Zobel, C. Simoni, D. Piazza, X. Nunez, D. Rodriguez, Business case and security requirements, Public Deliverable D5.1.1, S3MS. <http://s3ms.org>, October 2006.