ConSpec: A Formal Language for Policy Specification

Irem Aktug
KTH
irem@nada.kth.se

Katsiaryna Naliuka
UNITN
naliuka@dit.unitn.it

1 Introduction

As mobile devices become increasingly popular, the problem of secure mobile application development gains importance. Mobile devices contain personal information, which users wish to protect. They also provide access to costly functionality, such as GSM services and GPRS connections. It is necessary to provide *controlled* access to the sensitive resources through fine-grained, at times application specific, constraints on execution.

A *security policy* specifies the set of acceptable executions and can thus be 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. A program adheres to a policy if all its executions are acceptable by the policy. Several techniques exist to ensure that an application complies to a policy. *Static verification* techniques analyze the program code in order to construct a mathematical proof that no execution of the program can violate the policy. Though such an analysis provides full assurance, due to the complexity of the problem, static verification is most often unfeasible in the resource-critical environment of the mobile device. *Runtime monitoring*, on the other hand, observes the behavior of a target program at runtime and terminates it if it does not respect the policy. Monitoring can effectively enforce many practically useful security policies [8]. However, it creates performance overhead since each *security relevant action* of the program should be detected and checked against the policy.

Our framework combines static with dynamic techniques in order to enforce security policies on mobile devices in the most effective way. It is similar to the *model-carrying code* approach [9], in that security specifications can be enforced at the three stages of the application lifecycle: the *development*, *installation* and *runtime* phases.

Development Phase: We associate with the application a *contract* [2], a piece of data that describes its *security-relevant behavior*. In the development phase, the contract is a specification of the intended security-relevant behavior of the application by the producer. The compliance of the application to the contract can be checked using static verification. The analysis can be performed by the producer or a trusted third party, which 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). Instead of signing with a private key, *proof-carrying code* techniques [7] can be used to convey assurance in program-contract compliance. If contract compliance can not be statically verified, then an execution monitor can be *inlined* in the program at this stage so that compliance is ensured at runtime [4, 3].

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; we call this process *contract-policy matching*. In the case where the contract does not match to the policy, adherence to policy can be enforced by inserting a monitor to the environment of the program in the installation phase.

Runtime: At runtime, the behavior of an application may be checked against a policy by monitoring.

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 in a framework such as the one described above. ConSpec aims for a balance of language expressiveness and tractability of the various tasks identified in the framework. For example, the problem of matching a contract against a policy reduces to the language containment problem for such automata, if policies and contracts are captured with automata on infinite strings. The complexity of this task (see e.g. [5, 6]) severely restricts the expressive power of the language. We provide a semantics for ConSpec and we briefly explain how the design of ConSpec renders possible the formal treatment of several of the activities in the framework mentioned above.

2 ConSpec Language

ConSpec is strongly inspired by the policy specification language PSLang, which was developed by Erlingsson and Schneider [3] for runtime monitoring. PSLang policies consist of a set of variable declarations representing the security state, 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 trades the formalisation of the monitor to the simplicity of inlining it. While a PSLang policy text is intended to encode a security automaton, a formal semantics for PSLang 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. Furthermore, contract-policy matching would be undecidable when such an expressive language is used. ConSpec is a more restricted language than PSLang: the domains of the security state variables are finite and 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 and elegant semantics. ConSpec has the additional scope construct for expressing security requirements on different levels. Case studies show that this feature is necessary for expressing many interesting real-life policies [10]. ConSpec is expressive enough to write policies on multiple executions of the same application, and on executions of all applications of a system, in addition to policies on a single execution of the application and on lifetimes of objects of a certain class.

Assume that the method `Open` of the class `File` is used to create files (when the argument `mode` has the value "CreateNew") or open files (when the argument `mode` has the value "Open") for reading or writing. Similarly, the method `Open` of the class `Connection` opens a connection and the method `AskConnect` asks the user for permission to open a connection. The policy "*Application must, after accessing an existing file, get approval from the user before opening a connection*" is expressed 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") -> { accessed = true; }


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

AFTER string answer = GUI.AskConnect()
PERFORM
      answer.equals("Yes") -> { permission = true; }
```

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). 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 (the second guard). In order to decide if the application is allowed to open a connection, history of existing file accesses and user permissions should be consulted. 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 has the value true at a certain point in the execution is, if the last execution of the method `GUI.AskConnect` has returned "Yes". 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. If neither of these conditions hold for the current call, that is if an existing file has been accessed but no permission for connection was granted, then it is a violating call as none of the guards are satisfied.

For such a language, an elegant semantics can be given in terms of security automata [8].

3 ConSpec in Use

Current work focuses on the formal treatment of the following tasks related with policy enforcement, based on ConSpec semantics:

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 and standard methods for checking language inclusion for automata can be facilitated for contract-policy matching (see for instance [1]).

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. 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 *before* action is immediately preceded by a transition of the automaton for the same action; dually, each *after* action is immediately followed by a corresponding automaton transition. It is simple, then, 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 [3] for details). The correctness of such a monitor inlining scheme 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 (induced by the policy). If the party responsible for the inlining is not trusted, the proof-carrying code approach can be used. An annotation scheme can be devised that produces annotations for a given policy, so that if the annotations are valid for a program, then the program adheres to the policy. The inlining party can use certain information about the inlining (e.g. the variables used to represent the security state) to prove the verification conditions resulting from annotating the program with this scheme. A proof can then be shipped to the consumer which enables the correctness of the inlining to be verified on the mobile device, based on the same scheme.

4 Conclusion

In this paper, we present the language ConSpec, which can be used for specifying both policies and contracts in various security enforcement related tasks of the application lifecycle. We provide a formal semantics for the language which enables formal proofs to be constructed for these taks. Currently, we are formalizing enforcement techniques using ConSpec for sequential programs as summarized in Section 3. As future work, we aim to extend our approach to applications where multiple threads can perform security-relevant actions.

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

References

[1] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288, 1992.

[2] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *European PKI Workshop: Theory and Practice (to appear)*, 2007.

[3] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Department of Computer Science, Cornell University, 2004.

[4] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of the Workshop on New Security Paradigms (NSPW '99)*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[5] J. E. Hopcroft. On the equivalence and containment problems for context-free languages. *Theory of Computing Systems*, 3(2):119–124, 1969.

[6] H. B. Hunt and D. J. Rosenkrantz. On equivalence and containment problems for formal languages. *J. ACM*, 24(3):387–396, 1977.

[7] G. C. Necula. Proof-carrying code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[8] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[9] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. *ACM SIGOPS Operating Systems Review*, 2003.

[10] A. Zobel, C. Simoni, D. Piazza, X. Nuez, and D. Rodriguez. Business case and security requirements. Public Deliverable of EU Research Project D5.1.1, S3MS- Security of Software and Services for Mobile Systems, http://s3ms.org, October 2006.