The Journal of Logic and Algebraic Programming 78 (2009) 340–358

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Matching in security-by-contract for mobile code☆

N. Bielova, N. Dragoni [1], F. Massacci [*], K. Naliuka, I. Siahaan

Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento, Italy

ARTICLE INFO

Available online 28 February 2009

Keywords:
Security-by-contract
Mobile code
Security
Run-time monitor

ABSTRACT

We propose the notion of *security-by-contract*, a mobile contract that an application carries with itself. The key idea of the framework is that a digital signature should not just certify the origin of the code but rather bind together the code with a contract.
We provide a description of the workflow for the deployment and execution of mobile code in the setting of security-by-contract, describe a structure for a contractual language and propose a number of algorithms for one of the key steps in the process, the *contract-policy matching* issue. We also describe the prototype for matching policies with security claims of mobile applications that we have currently implemented.
We argue that security-by-contract would provide a semantics for digital signatures on mobile code thus being a step in the transition from trusted code to trustworthy code.

1. Introduction

Mobile devices are increasingly popular and powerful. Yet, the growth in computing power of nomadic devices has not been supported by a comparable growth in available software: on high-end mobile phones we cannot even remotely find the amount of third party software that was available on our old PC.

One of the reasons for this lack of applications is also the security model adopted for mobile phones. The current security model is exemplified by the JAVA MIDP 2.0 and .NET CF approach and is based on trust relationships: a mobile application is accepted if it is digitally signed by a trusted party. The level of trust of the "trusted party" determines the privileges of the code by essentially segregating it into appropriate trust domain.

The problem with trust relationship, i.e. digital signatures on mobile code, is twofold. At first we can only reject or accept the signature. This means that inter-operability in a domain is either total or not existing: an application from a not-so-trusted source can be denied network access, but it cannot be denied access to a specific protocol, or to a specific domain. For example, if a payment service is available on the platform and an application for paying parking meters is loaded, the user cannot block the application from performing large payments.

The second (and major) problem is that *there is no semantics attached to the signature*. The signature only tells us who is responsible for the code. This is a problem for both code producers and consumers.

From the point of view of mobile code consumers they must essentially accept the code "as-is" without the possibility of making informed decisions [41]. One might well trust SuperGame Inc. to provide excellent games and yet might decide to rule out games that keep playing while the battery fells below 20%. At present such choice is not possible.

From the point of view of the code producer they produce code with unbounded liability. They cannot declare which security actions the code will do, they only declare that the code comes from their software factory. The consequence is

☆ This work has been partly supported by the Projects EU-IST-STREP-S3MS and EU-IST-IP-MASTER.
* Corresponding author. Tel.: +39 0461 882086.
E-mail addresses: ndra@imm.dtu.dk (N. Dragoni), fabio.massacci@unitn.it (F. Massacci).
[1] Current address: Department of Informatics and Mathematical Modeling, Technical University of Denmark, Denmark.

1567-8326/$ - see front matter © 2009 Published by Elsevier Inc.
doi:10.1016/j.jlap.2009.02.013

that injecting an application into the mobile market is a costly operation as SME developers must essentially convince the operators that their code will not do anything harmful. As a result, a lot of code comes at the market uncertified or self-certified. In other words, most code is untrusted.

To deal with the untrusted code the mobile version of either .NET [30] or Java [18] enables devices to exploit the mechanism of permissions. The drawback of permissions is that after assigning a permission the user has very limited control over how the permission is used. An application can receive a permission to send SMS messages and then send hundreds of them invisibly for the user. Once again the consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they are trusted (and then they can do almost everything).

Such situation has essentially polarized users in two groups: the security paranoids and the technology enthusiasts. From the former perspective[2]:

> The policy should protect the integrity of the device, and of other applications on the device, from any application that is loaded, i.e. sandboxing. [...] If the agenda data is sensitive, then I NEVER want untrusted applications to access it. This is much simpler than a temporal requirement that an untrusted application cannot have network access if it has looked at sensitive data.

The life of the technology enthusiasts is riskier but surely better[3]:

> There is this nice midlet that accesses my agenda and at 7:50 pops up a window that today is the birthday of these people in the list. [...] I'm no longer the only parent who never greets my contact children's teachers on their birthday. Look, there is also an option to send an SMS with happy birthday to the phone numbers but that would be too expensive with my current subscription.

Unfortunately, the technology enthusiast cannot just say that access to the agenda is fine provided there is no network connection. If the permission is granted the application may do anything with the obtained information, including sending it to a hackers' web site in Russia. If the required permission is not granted the application becomes completely useless. We need something beyond sandboxing.

1.1. The contribution of the paper

We propose in this paper the notion of *Security-by-Contract* (S×C) (as in programming-by-contract [35,7]) namely, the digital signature should not just certify the origin of the code but rather bind together the code with a contract. Loosely speaking, a *contract* contains a description of the relevant features of the application and the relevant interactions with its host platform. A mobile platform could specify platform contractual requirements, a *policy*,[4] which should be matched by the application's contract. Among the relevant features, one can list fine-grained resource control (e.g. silently initiate a phone call or send a SMS), memory usage, secure and insecure web connections, user privacy protection, confidentiality of application data, constraints on access from other applications already on the platform.

We provide here a description of the overall life-cycle of mobile code in the setting of security-by-contract, describe a structure for a contractual language and propose a number of algorithms for one of the key steps in the process, namely the issue of *contract-policy matching*.

We argue that security-by-contract would provide a semantics for digital signatures on mobile code thus being a step in the transition from trusted code to trustworthy code.

In the next section we briefly describe the basic idea of the S×C paradigm for mobile code and security policies in the Security-By-Contract scenario. Then we discuss a concrete contractual language (Section 3) that we have used. The subsequent three sections describe the matching algorithm structure and its components. The software architecture and our Java implementations are then described (Section 7). A discussion of related work and some conclusions end the paper (Section 8).

2. The S×C usage model

Bertrand Meyer in "The Grand Challenge of Trusted Components" at ICSE 2003, described the challenges to be met by designers of trusted software components. A key challenge was the lack of an explicit specification of the relevant behavior of the components and, above all, the development of computer aided methods for checking and enforcing those explicit specifications at run-time.

In his EIFFEL project, Meyer suggested the notion of programming-by-contract [7,22] as a way to address this problem: software components should come with a contract described in contractual aware languages, the correctness of a program is explicitly asserted through a contract, i.e. a claim on the behavior of single methods (pre-conditions and post-conditions) or the whole classes.

Applications wishing to use a component developed with programming-by-contract can then match the contractual claims by the component with their need or preferences. The trusted component is bound by the promises it makes thus implicitly stipulating a contract with the applications that wishes to uses its services.

[2] A reviewer at a leading security conference on a paper on usage control.
[3] An article on the local press commenting the features of the "Andiamo" (let's go) student software project.
[4] In the sequel we will refer to policy as the security requirements on the platform side and by contract the security claims made by the mobile code.

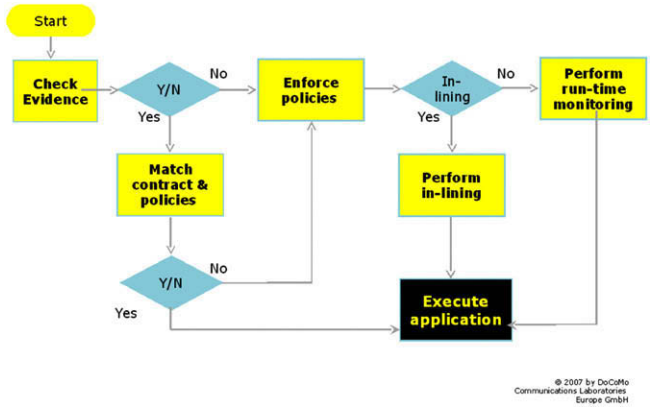


Fig. 1. SxC workflow.

In the framework of the S3MS project (www.s3ms.org), we have proposed to apply Meyer's intuition to security claims of mobile code. A contract is an explicit claim on the security behavior of a mobile application (playing the role of Meyer's trusted component) that a platform should match against its own.

Such notion fills a gap in the current arena of security mechanisms for enforcing security properties of mobile code. The focus of run-time security monitors [13,3,20,48] is to enforce a security policy specified by the platform. The possibility that an application can make security claims about its behavior is not considered. We thus need to monitor also applications that are harmless. At the other side of the spectrum of security research, the proof-carrying code approach [39] produces a "certificate", a machine checkable proof that the object code respects a given security policy. The most basic example is SUN's javac compiler: it takes a Java source code that satisfies a type safety policy and produces JVML code that obeys type-safety. This approach is restricted to type-safety and does not allow the application to claim some ad-hoc security behavior: "no http connection to Google Analytics" would be a simple claim that could not be checked with traditional language-based approaches.

The mobile code developers are responsible to provide a description of the security behavior that their code provides.

Definition 2.1 (*Contract*). A contract is a formal complete and correct specification of the behavior of an application for what concerns relevant security actions (virtual machine API Calls, Operating System Calls).

Following Meyer's intuition, the contract has to be complete so that all operations that are security-relevant and are used in the application should be included in the contract. Further, it should be correct in the sense that the behavioral specification of the security actions described in the contract should be the one actually followed in the application. In this setting we found useful to provide both rules that describe the behavior during a single run of the application (a session) and rules that describe properties across multiple run of the application.

At *development time* the mobile code developers are responsible for providing the description of the contract. Such pair of contract and mobile code may also undergo a formal certification process by the developer's own company, the smart card provider, a mobile phone operator, or any other third party for which the application has been developed. By using suitable techniques such as static analysis [27,46], certified monitor in-lining [19,48,9], or general theorem proving, the code is certified to comply with the developer's contract.

At the end of the process, the code and the security claims are sealed together with the evidence for compliance (either a digital signature or a PCC proof [39,9]) and shipped for deployment. By signing the code and the contract together the developer certifies that the code complies with the stated claims on its security-relevant behavior.

On the other side we can see that users and mobile phone operators are interested in all codes that are deployed on their platform to be secure according their priorities. In other words they must declare their security policy:

Definition 2.2 (*Policy*). A policy is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions (Virtual Machine API Calls, Operating System Calls).

At *deployment time*, the target platform follows a workflow similar to the one depicted in Fig. 1 (see also [49] for the details of the software architecture). First, it checks that the evidence is correct. As we said already, such evidence can be a trusted signature as in standard mobile applications [51] or a proof that the code satisfies the contract (and then one can use PCC techniques to check it [39]).

Once we have evidence that the contract is trustworthy, the platform checks, that the claimed policy is compliant with the policy that our platform wants to enforce. If it is, then the application can be run without further ado. This may be a significant saving from in-lining a security monitor or actually deploying a run-time monitor in parallel with the application.

At *run-time* we might want to decide to still monitor the application. Then, we might decide to inline a number of checks into the application so that any undesired behavior can be immediately stopped or corrected.

To give an intuitive idea of the constructs proposed in this paper we informally describe them by means of rules in natural language. We will formally specify the rules in Section 3. Let us consider the following example which we will use throughout the paper.

Example 2.1. A midlet[5] claims that during a single execution it will
1. only use HTTPS network connections and
2. send no SMS messages.

The platform's policy has two rules applicable to a single run of the application:
1. the application uses only high-level (HTTP, HTTPS) network connections;
2. a maximum of five text messages can be sent by the application.

It should be intuitive that in this case the application's contract matches the platform's policy. In fact, the security behavior claimed in the application's contract corresponds to the allowed security behavior stated in the platform's policy.

2.1. Contract-policy matching

As we can see in Fig. 1, one of the key problems in the overall security-by-contract workflow is *contract-policy matching* which is also a key aspect in Meyer's programming-by-contract: *given a contract that an application carries with itself and a policy that a platform specifies, is the contract compliant with the policy?*

Contract-policy matching represents a common problem in the life-cycle because it must be done at all levels: both for development and run-time operation. Intuitively, matching should succeed if and only if by executing the application on the platform every behavior of the application that satisfies its contract also satisfies the platform's policy. To address this issue we need efficient algorithms to match application contracts with device policies. This will be the target of Section 4.

The following example shows how matching might fail.

Example 2.2. Let us consider an application's contract with just one rule:
- the amount of data the application can receive is bounded by 1024 kb.

The platform's policy has one rule allowing to receive a specific amount of data.
- the amount of data an application can receive is bounded by 512 kb.

In this case, the contract-policy matching must fail, since the application can receive more data than the one allowed by the mobile platform.

3. A concrete contract representation

If a contract represents the security behavior of an application the temptation would be to make such contractual claims arbitrarily complex. Since we argue that contract should be matched by mobile devices a complex procedure is likely to defy the very spirit of our proposal.

Further, a number of independent security requirements analyses for mobile and distributed systems [24,37,52] show that detailed contracts are not really necessary. The characteristic feature of these applications is that they need wide access to services to execute correctly. Yet, the user still wants to control that these services are not abused or misused. Therefore the same permission can be granted or not granted depending, for instance, on previous actions of the midlet or some conditions on application environment.

Some examples of security policies for mobile devices include:
1. The application sends no more than a specific number of messages in each session.
2. The application only loads each image from the network once.
3. The delay between two periodic invocations of the midlet is at least T.
4. The application does not initiate calls to international numbers.
5. The application only uses files whose names match a given pattern.
6. The application does not send MMS messages.
7. The application connects only to its origin domain.
8. The application does not use the *FileConnection.delete()* function.
9. The application only receives SMS messages on a specific port.
10. The length of a SMS message sent does not exceed the payload of a single SMS message.
11. The application must close all files that it opens.

[5] A Java application that conforms to the Mobile Information Device Profile (MIDP) standard.

```
MAXINT MaxIntValue
MAXLEN MaxLenValue
RuleID Identifier

SCOPE <Object ClassName | Session | MultiSession
    | Global>

SECURITY STATE
    [CONST] | <bool | int | string>
                        VarName1 = <DefaultValue1>
    | <int> VarName2 = <DefaultValue2>
                        RANGE <FromValue> ..  <ToValue>
                        ...

<BEFORE | AFTER | EXCEPTIONAL> EVENT MethodSignature1
    PERFORM
    condition1 -> action1
                ...
    conditionM1 | ELSE> -> actionM1
                            ...
<BEFORE | AFTER | EXCEPTIONAL> EVENT MethodSignatureK
    PERFORM
    condition1 -> action1
                ...
    conditionMK | ELSE> -> actionMK
```

Fig. 2. A fragment of the ConSpec syntax.

Notice the difference between policies 1 and 2. The first one specifies the constraint on a single execution (*session*) of the program. The second one puts a restriction on all runs of the application. Policy 3 also requires to make a distinction between multiple sessions of the application. For this reason, the contract must include the constructs that define the *scope* of the obligation. Moreover, such policies as policy 11 are most naturally expressed at the level of separate objects (in this case objects of type *FileConnection*).

We provide an overview of the ConSpec syntax, the language exploited to specify contracts and policies within the context of the S×C framework. A full description of the language is outside the scope of the paper (interested readers can consult [2]).

A specification in ConSpec is a non-empty list of rules. Each rule is defined for the specific part of contract (e.g. rule for the SMS messages, for Bluetooth connections, etc.) and describes security properties for the given part. Fig. 2 shows a fragment of the ConSpec syntax for specifying one single rule.

The `RuleID` tag identifies the area of the contract, e.g. for restriction of sending text messages the identifier could be `"TEXT_MESSAGES"` or for accessing the file system the identifier could be `"FILE_ACCESS"`.

Each rule consists of three parts: scope definition, state declaration and a list of event clauses.

There are different scopes in ConSpec: scope `Object` is used when the rule can be applied for the object of specific class; scope `Session` if the security properties are applicable for the single run of the application; scope `Multisession` when the rule describes behavior of the application during its multiple runs and scope `Global` for executions of all applications of a system. This notion of scope definition in ConSpec differs from scope in MIDP because the scope in MIDP defines set of capabilities, namely User interface, Persistent storage, Networking, and Timers.

Example 3.1. Fig. 3 shows the ConSpec specifications of contract and policy of Example 2.1. Both specifications consist of two rules, one with the identifier `HIGH_LEVEL_CONNECTIONS` and the other one `SMS_MESSAGES`. In each case the first rule specifies the restrictions on using the data connections, and the second one on sending short messages. Each identifier of the rule is followed by the scope declaration. As the policy is to be applied to each run of the application separately, the appropriate scope is `Session` (which is, in fact, the most common one).

The state space of the policy is defined after the scope. The state declaration defines the state variables to be used in the current rule of the ConSpec specification. The variables can be constant and non-constant. All non-constant variables characterize the state of the automaton defined by the rule. Constant variables are simply used in the specification and do not play significant role in automaton construction.

Variables can be boolean, integer or string. However, the states have to be finite and all types have to be bounded. Hence, a ConSpec specification has two tags: `MAXINT` to define maximum value of integers and `MAXLEN` to define maximum length of strings. If variables should have less variability, then the keyword `RANGE` is used for a more precise bound, as in rule `SMS_MESSAGES` of the policy (Fig. 3b).

The most important part of the rule is a sequence of event clauses. The event clauses define the transitions of the automaton constructed from the ConSpec rule. Each event clause has a list of guard conditions and an update block which will be performed when the corresponding guard condition holds.

```
MAXINT 10000 MAXLEN 10
RULEID HIGH_LEVEL_CONNECTIONS

SCOPE Session

SECURITY STATE

BEFORE javax.microedition.io.Connector.open(string url) PERFORM
  url.startsWith("https://") -> {skip;}


RULEID SMS_MESSAGES
SCOPE Session

SECURITY STATE

BEFORE javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TextMessage msg) PERFORM
  false -> {skip;}

AFTER javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TestMessage msg) PERFORM
  false -> {skip;}
```

(a) ConSpec Specification of the Contract of Example 2.1

```
MAXINT 10000 MAXLEN 10
RULEID HIGH_LEVEL_CONNECTIONS

SCOPE Session

SECURITY STATE

BEFORE javax.microedition.io.Connector.open(string url) PERFORM
  (url.startsWith("http://") || url.startsWith("https://")) -> {skip;}


RULEID SMS_MESSAGES
SCOPE Session

SECURITY STATE
CONST int maxMessage = 5;
int messageSent = 0 RANGE 0..5;

BEFORE javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TextMessage msg) PERFORM
  messageSent < maxMessage -> {skip;}

AFTER javax.wireless.messaging.MessageConnection.send
(javax.wireless.messaging.TextMessage msg) PERFORM
  true -> {messageSent = messageSent + 1;}
```

(b) ConSpec Specification of the Policy of Example 2.1

Fig. 3. ConSpec specification of the contract and the policy of Example 2.1.

Every event is defined by a modifier and a signature API method, including name of the class, method name and optionally list of parameters. The modifiers (`BEFORE`, `AFTER` and `EXCEPTIONAL`) indicate the moment, in which the update block must be executed.

Example 3.2. In the first rule (`HIGH_LEVEL_CONNECTIONS`) of the contract and the policy from Example 2.1 only one event is raised each time before the method to open a connection (`javax.microedition.io.Connector.open`) is called.

When the program attempts to call the security-relevant method the guard conditions are evaluated in the same order, in which they appear in the policy. The condition is a boolean expression on the state variables and possible parameters of the method. If the condition evaluates to true then the corresponding update block is executed, and the control returns to the target program.

Example 3.3. In our contract, if the program attempts to call the method `javax.microedition.io.Connector.open` with the `url` starting with *https* the method is executed without performing any update (the keyword `skip` denotes the empty update block). Otherwise if the `url` string does not start with *https* (the attempted connection is not secure) no condition evaluates to *true*, so the contract is violated.

```
MAXINT 10000 MAXLEN 10
RULEID LIMITED_DATA

SCOPE Session

SECURITY STATE
CONST int maxKbRecieve = 1024;

BEFORE System.Net.Sockets.BeginReceive (Byte[] buffer,
int offset, int size, System.Net.Sockets.SocketFlags socketFlags,
System.AsyncCallback callback, Object state) PERFORM
    size < maxKbRecieve -> {skip;}
```

(a) ConSpec Specification of the Contract of Example 2.2

```
MAXINT 10000 MAXLEN 10
RULEID LIMITED_DATA

SCOPE Session

SECURITY STATE
CONST int maxKbRecieve = 512;

BEFORE System.Net.Sockets.BeginReceive(Byte[] buffer, int offset,
int size, System.Net.Sockets.SocketFlags socketFlags,
System.AsyncCallback callback, Object state) PERFORM
    size < maxKbRecieve ->{skip;}
```

(b) ConSpec Specification of the Policy of Example 2.2

Fig. 4. ConSpec Specification of the contract and the policy of Example 2.2.

The guard condition can be replaced by the keyword `ELSE`; in this case the corresponding update block will always run if all other blocks evaluate to false. If the condition is set to *false*, then the current event can never run according to the specification.

Example 3.4. In the second rule (`SMS_MESSAGES`) of the contract and the policy of Example 2.1 we define two events: before and after the method that sends a message. All security-relevant events that might be declared in the policy should be presented in the contract, otherwise the contract-policy matching will fail when it should succeed. Therefore we need to declare `AFTER javax.wireless.messaging.MessageConnection.send` event in the contract in order to run the matching procedure on many different policies and have a correct result.

Example 3.5. Fig. 4 shows the ConSpec specifications of the contract and the policy of Example 2.2, respectively.

4. Contract-policy matching

In this section we provide a generic algorithm for contract-policy matching. The algorithm is *generic* since it does not depend on the formal model adopted for specifying the semantics of rules (process algebra, security automata, Petri Nets, and so on), but instead it is defined by means of specific abstract constructs. Therefore, to exploit the algorithm it will be sufficient to have an implementation of these constructs in the formal language adopted for specifying rules. In Section 6 we will provide an automata-based implementation of such constructs, giving in this way a complete version of the algorithm for rules formally specified with finite-state automata.

We have identified the following abstract operators (C and P indicate a generic contract and policy respectively):

- [Combine Operator $\oplus$] $\text{Spec} = \oplus_{i=1,\ldots,n}\text{Spec}_i$
  It combines all the formal rule specifications $\text{Spec}_1, \ldots, \text{Spec}_n$ in a new specification Spec.
- [Simulate Operator $\approx$] $\text{Spec}^C \approx \text{Spec}^P$
  It returns 1 if formal rule specification Spec^C can be followed accordingly by formal rule specification Spec^P, 0 otherwise. Intuitively, if the policy cannot follow then the contract is not compliant, i.e. that particular rule is a violation.
- [Contained-By Operator $\sqsubseteq$] $\text{Spec}^C \sqsubseteq \text{Spec}^P$
  It returns 1 if the behavior specified by Spec^C is among the behaviors that are allowed by Spec^P, 0 otherwise.
- [Traces Operator] $\mathcal{S} = \text{Traces}(\mathit{Spec})$
  It returns the set $\mathcal{S}$ of all the possible sequences of actions that can be performed according to the formal specification *Spec*.

We assume that the above abstract constructs are characterized by the following properties:

Property 4.1. $\mathit{Traces}(\mathit{Spec}_1 \oplus \mathit{Spec}_2) = \mathit{Traces}(\mathit{Spec}_1) \cup \mathit{Traces}(\mathit{Spec}_2)$.

Property 4.2. $\mathit{Spec}_1 \sqsubseteq \mathit{Spec}_2 \Leftrightarrow \mathit{Traces}(\mathit{Spec}_1) \subseteq \mathit{Traces}(\mathit{Spec}_2)$.

Property 4.3. $\mathit{Spec}_1 \approx \mathit{Spec}_2 \Rightarrow \mathit{Traces}(\mathit{Spec}_1) \subseteq \mathit{Traces}(\mathit{Spec}_2)$.

Definition 4.1 (*Exact Matching*). Matching should succeed if and only if by executing the application on the platform every trace that satisfies the application's contract also satisfies the platform's policy

$$\mathsf{Traces}\left(\oplus_{i=1,\ldots,n}\mathrm{Spec}_i^C\right) \subseteq \mathsf{Traces}\left(\oplus_{i=1,\ldots,m}\mathrm{Spec}_i^P\right)$$

Definition 4.2 (*Sound Sufficient Matching*). Matching should fail if by executing the application on the platform there might be an application trace that satisfies the contract and does not satisfy the policy.

Definition 4.3 (*Complete Matching*). Matching should succeed if by executing the application on the platform every trace satisfying the contract also satisfies the policy.

By applying Definition 4.2 we might reject "good" applications that are however too difficult or too complex to perform. On the other hand, Definition 4.3 may allow "bad" (malicious) applications to run but it will certainly accept all "good" ones (and "bad" applications can later be detected, for instance, by run-time monitoring). Examples of matching between contracts and policies follow. As shown in Fig. 5, the generic contract/policy matching algorithm takes as inputs two rule sets $\mathcal{R}^C$ and $\mathcal{R}^P$ representing respectively the contract and the policy to be matched. The algorithm checks if $\mathcal{R}^C$ "matches" $\mathcal{R}^P$.

Algorithm 1 lists the source code of the MatchContracts function, which is the root function of the whole algorithm. Basically, the algorithm works as follows. First of all, both rule sets $\mathcal{R}^C$ and $\mathcal{R}^P$ are partitioned according to the scope of the rules (lines 1 and 2). This is done by calling the Partition procedure (Algorithm 2) that partitions a generic rule set $\mathcal{R}$ in a sequence of rule sets with the same scope: $\left\langle \mathcal{R}_{SESSION}, \mathcal{R}_{MULTISESSION}, \{\mathcal{R}_{class}\}_{class \in \xi^C} \right\rangle$. This partition is necessary because in the S×C framework comparison of rules starts only within a certain scope. Once created two sequences of scope-specific rule sets (one for the contract and one for the policy), the algorithm checks if each rule set in the sequence of the contract matches the corresponding rule set in the sequence of the policy (lines 3–11). In other words, we match rules within the same scope. This is done by calling the MatchRules function (lines 4–6) that we discuss in the next paragraph. If all succeeds (line 11), then the contract matches the policy. Otherwise, matching fails.

5. Rule and specification matching

Matching between rules is performed by the MatchRules function (Algorithm 3). Since the rules of the two input sets $\mathcal{R}^C$ and $\mathcal{R}^P$ must have the same scope, before doing matching checks the algorithm cleans $\mathcal{R}^C$ and $\mathcal{R}^P$ removing the scope from each rule. As a consequence, two sets L^C and L^P of pairs $\left(\mathrm{ID}^{C/P}, \mathrm{Spec}^{C/P}\right)$ are built. Now the algorithm is ready to check the contract/policy match. Each pair in L^P is compared with the set L^C by means of the MatchSpec function (line 4). When a match is not found for a pair (line 6), i.e. the MatchSpec function returns 0, that pair is stored in a rule set L^P_{failed} (line 7).

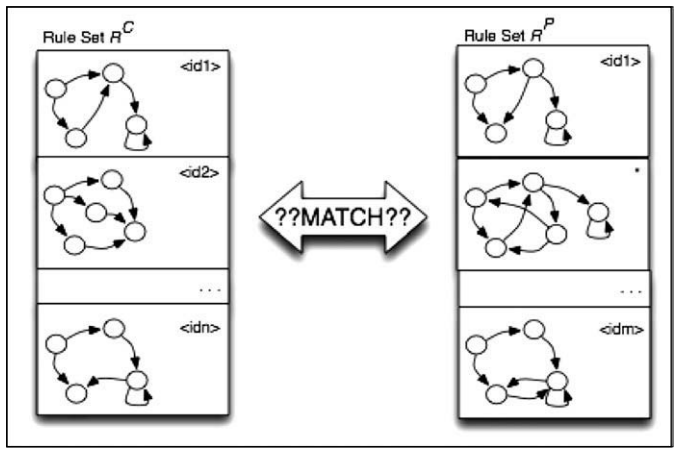


Fig. 5. Contract-policy matching problem.

If for all rules in L^P there exists a match with L^C, i.e. the MatchSpec function returns 1 for each pair in L^P so that $L^P_{failed} = \emptyset$, then the match between rules succeeds and the algorithm returns 1 (lines 10–11). Otherwise, if $L^P_{failed} \neq \emptyset$ (i.e. there are no rules in L^C that match with the rules of L^P_{failed}) then the algorithm performs a last "global" check. More precisely, the combination of the rules in L^C is matched with the combination of the rules in L^P_{failed} (line 13). If also this match does not succeed, then the algorithm returns 0, otherwise it returns 1.

Algorithm 1 MatchContracts Function

Input: rule set $\mathcal{R}^C$, rule set $\mathcal{R}^P$
Output: 1 if $\mathcal{R}^C$ matches $\mathcal{R}^P$, 0 otherwise
1: $\left\langle \mathcal{R}^C_{SESSION}, \mathcal{R}^C_{MULTISESSION}, \left\{\mathcal{R}^C_{class}\right\}_{class\in\zeta^C} \right\rangle \Leftarrow \mathsf{Partition}\left(\mathcal{R}^C\right)$
2: $\left\langle \mathcal{R}^P_{SESSION}, \mathcal{R}^P_{MULTISESSION}, \left\{\mathcal{R}^P_{class}\right\}_{class\in\zeta^P} \right\rangle \Leftarrow \mathsf{Partition}\left(\mathcal{R}^P\right)$
3: **if** $\mathsf{MatchRules}\left(\mathcal{R}^C_{SESSION}, \mathcal{R}^P_{SESSION}\right)$ **then**
4: **if** $\mathsf{MatchRules}\left(\mathcal{R}^C_{MULTISESSION}, \mathcal{R}^P_{MULTISESSION}\right)$ **then**
5: **for all** *class* $\in \zeta^P$ **do** // *for all classes in policy*
6: **if** $\mathsf{MatchRules}\left(\mathcal{R}^C_{class}, \mathcal{R}^P_{class}\right)$ **then** // *if class* $\notin \zeta^C$, *then* $\mathcal{R}^C_{class} = \emptyset$
7: skip
8: **else**
9: return(0)
10: **end if**
11: **end for**
12: return(1)
13: **end if**
14: **end if**
15: return(0)

Algorithm 2 Partition Procedure

Input: rule set $\mathcal{R}$
Output: $\left\langle \mathcal{R}_{SESSION}, \mathcal{R}_{MULTISESSION}, \{\mathcal{R}_{class}\}_{class\in\zeta} \right\rangle$
1: $\mathcal{R}_{SESSION} \Leftarrow \{r \in \mathcal{R} \mid \mathrm{Scope}(r) = \mathrm{SESSION}\}$
2: $\mathcal{R}_{MULTISESSION} \Leftarrow \{r \in \mathcal{R} \mid \mathrm{Scope}(r) = \mathrm{MULTISESSION}\}$
3: **for all** *class* $\in \zeta$ **do** // *for all classes in contract/policy*
4: $\mathcal{R}_{class} \Leftarrow \{r \in \mathcal{R} \mid \mathrm{Scope}(r) = \mathrm{OBJECT} < class >\}$
5: **end for**

Algorithm 3 MatchRules Function

Input: rule set $\mathcal{R}^C$, rule set $\mathcal{R}^P$
Output: 1 if $\mathcal{R}^C$ matches $\mathcal{R}^P$, 0 otherwise
1: $L^C \Leftarrow \left\{ \left(\mathrm{ID}^C,\ \mathrm{Spec}^C\right) \;\middle|\; \left\langle \mathrm{scope}, \mathrm{ID}^C, \mathrm{Spec}^C \right\rangle \in \mathcal{R}^C \right\}$
2: $L^P \Leftarrow \left\{ \left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right) \;\middle|\; \left\langle \mathrm{scope}, \mathrm{ID}^P, \mathrm{Spec}^P \right\rangle \in \mathcal{R}^P \right\}$
3: **for all** $\left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right) \in L^P$ **do**
4: **if** $\mathsf{MatchSpec}\left(L^C, \left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)\right)$ **then**
5: skip
6: **else** // *may return* $\emptyset$ *for efficiency*
7: $L^P_{failed} \Leftarrow L^P_{failed} \cup \left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)$
8: **end if**
9: **end for**
10: **if** $L^P_{failed} = \emptyset$ **then**
11: return(1)
12: **else**
13: return$\left(\mathsf{MatchSpec}\left(\left(*,\ \oplus_{(\mathrm{ID}^C,\ \mathrm{Spec}^C)\in L^C}\right), \left(*,\ \oplus_{(\mathrm{ID}^P,\ \mathrm{Spec}^P)\in L^P_{failed}}\right)\right)\right)$
14: **end if**

Algorithm 4 MatchSpec Function

Input: $L^C = \left\langle \left(\mathrm{ID}_1^C,\ \mathrm{Spec}_1^C\right), \ldots, \left(\mathrm{ID}_n^C,\ \mathrm{Spec}_n^C\right)\right\rangle, \left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)$

Output: 1 if L^C matches $\left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)$, 0 otherwise

1: **if** $\exists \left(\mathrm{ID}^C,\ \mathrm{Spec}^C\right) \in L^C \wedge\ \mathrm{ID}^C = \mathrm{ID}^P$ **then**
2: **if** HASH(Spec^C) = HASH(Spec^P) **then**
3: return(1)
4: **else if** $\mathrm{Spec}^C \approx \mathrm{Spec}^P$ **then**
5: return(1)
6: **else if** $\mathrm{Spec}^C \sqsubseteq \mathrm{Spec}^P$ **then**
7: return(1)
8: **else** // *Restriction: if same ID then same specification must match*
9: return(0)
10: **end if**
11: **else**
12: MatchSpec$\left(\left(*,\ \oplus_{(\mathrm{ID}^C,\ \mathrm{Spec}^C)\in L^C}\right), \left(*,\ \mathrm{Spec}^P\right)\right)$
13: **end if**

The MatchSpec function (Algorithm 4) checks the match between a set of pairs $\mathcal{L}^C = \left\langle \left(\mathrm{ID}_1^C,\ \mathrm{Spec}_1^C\right), \ldots, \left(\mathrm{ID}_n^C,\ \mathrm{Spec}_n^C\right)\right\rangle$ and a pair $\left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)$ representing respectively the rules of the contract and a rule of the policy to be matched. The function returns 1 in two situations:

1. there exists a pair $\left(\mathrm{ID}^C,\ \mathrm{Spec}^C\right)$ in L^C that matches with $\left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)$
2. the combination of all the specifications in L^C matches with $\left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)$

Otherwise, the function returns 0.

Specification matching is verified as follows. If there exists a pair $\left(\mathrm{ID}^C,\ \mathrm{Spec}^C\right)$ in L^C such that ID^C is equal to ID^P (line 1), then the algorithm checks the hash values of the specifications Spec^C and Spec^P. Matching succeeds if they have the same value (line 2). Otherwise, the algorithm checks if Spec^C simulates Spec^P (line 4). If this is the case, then the matching succeeds, otherwise the more computationally expensive containment check is performed (line 6). If also this check fails, the algorithm ends and matching fails (because the rules with the same ID must have the same specification).

If there exists no pair in L^C such that ID^C is equal to ID^P (line 11) then the algorithm checks the match between the combination of all the specifications in L^C and $\left(\mathrm{ID}^P,\ \mathrm{Spec}^P\right)$ (line 12).

It should be clear now why the described algorithm is generic. To be actually executed, it needs a well defined formal semantics and implementation of the abstract constructs. Several formal tools might be used for this purpose, such as standard process algebras, Petri Nets and so on. As a concrete example, in the next Section we will discuss the case of rules specified by means of specific automata, providing a real implementation of the constructs and as a result of the overall matching algorithm.

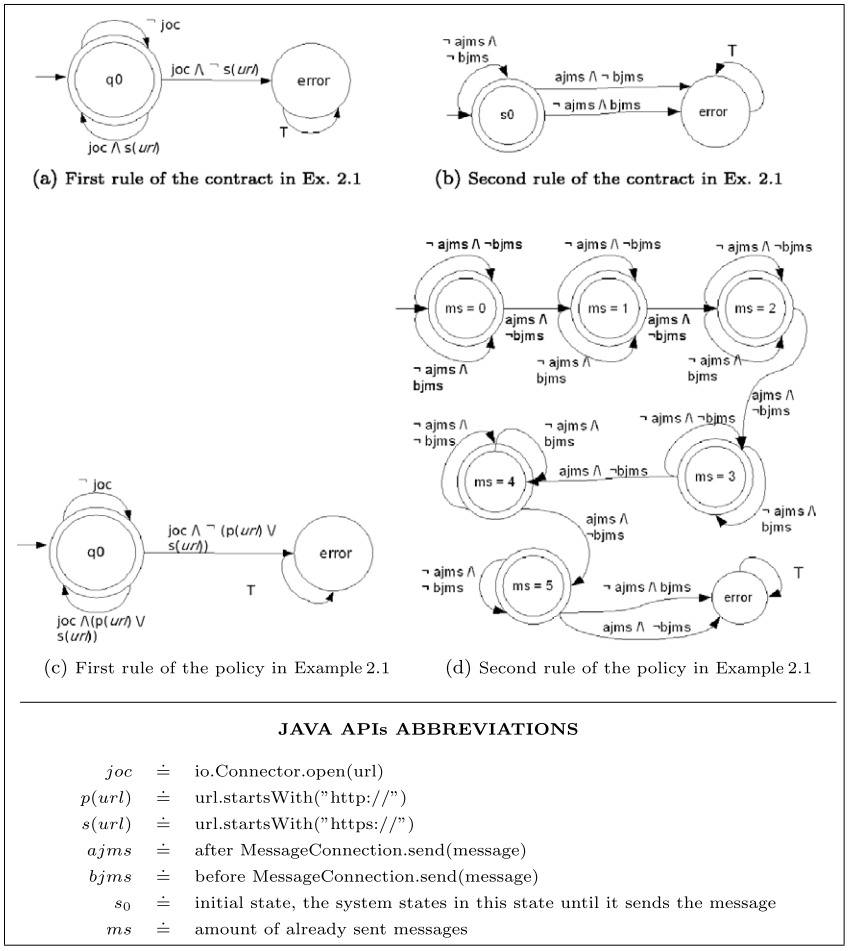
6. Contract-policy matching with security automata

In this section we show how the matching algorithm can be used when the behavior of rules is specified by means of specific automata, namely *Automata Modulo Theory* ($\mathcal{AMT}$) [33]. As already remarked in Section 4 we just need to provide an implementation of the $\oplus$, $\sqsubseteq$ and $\approx$ operators used in Algorithms 3 and 4. For the sake of clarity, we briefly introduce $\mathcal{AMT}$. Then we briefly describe the algorithms for implementing the abstract constructs.

$\mathcal{AMT}$ generalizes the finite state automata of model-carrying code [45] and extends Büchi Automata (BA). The theory of $\mathcal{AMT}$ is a combination of the theory of Büchi Automata (BA) with satisfiability-modulo-theory (SMT) problem. SMT problem, which decides the satisfiability of first-order formulas modulo background theories, stretch the boundaries of formal verification based on effective SAT solvers. In contrast to classical security automata we prefer to use BA because, besides safety properties, there are also some liveness properties which have to be verified. An example of liveness is "The application uses all the permissions it requests".

$\mathcal{AMT}$ automaton which represents a model of the system can be extracted directly from the control-flow graph of the program. This automaton specifies *actual behavior* of the system. An automaton that specifies the *desired behavior* can be either built directly or from other specification languages. For example, a finite state automaton for a temporal logic specification can be constructed using the tableaux method [29].

Example 6.1. To illustrate how $\mathcal{AMT}$ can be used for property specification let us show the automata for the contract and the policy in Example 2.1 represented in Fig. 6. The automaton on Fig. 6a represents an automaton for the first rule of contract. Starting from state q_0 we stay in this state while no connection is opened ($\neg joc$). When a connection is opened, we stay in



(a) First rule of the contract in Ex. 2.1 (b) Second rule of the contract in Ex. 2.1

(c) First rule of the policy in Example 2.1 (d) Second rule of the policy in Example 2.1

Fig. 6. Automata specifying the rules of the contract and the policy of Example 2.1.

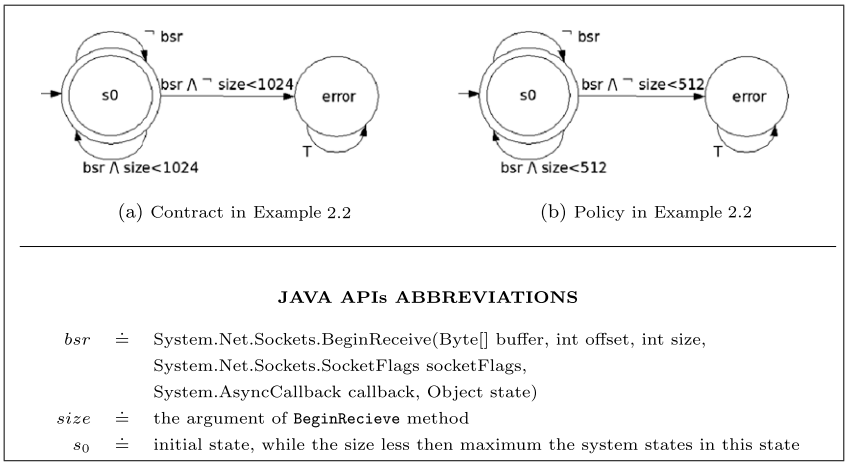
this state only if started connection Connector.open(string url) method is a secure one, i.e. url starts with "https://". We enter state *error* if we start an unsecure connection Connector.open(string url), e.g. url starts with "http://" or "sms://" etc. In the automaton for the second rule of the contract, the conditions of the transactions that represent the change of a state after the occurrence of the security relevant event have the form $ajms \wedge \neg bjms$ or $bjms \wedge \neg ajms$ instead of more simple *ajms* or *bjms*. The purpose of this is to reflect faithfully the fact that in the program according to our assumption two security-relevant events can never occur in the same moment. These examples are from a Java VM. Since we do not consider useful to invent our own names for API calls we use the `javax.microedition` APIs (though a bit verbose) for the notation that is shown in Fig. 6.

The exploitation of automata for formally specifying rules allows a straightforward implementation of the combine operator $\oplus$: rules are combined by simply making the synchronous product of the related automata.

Also contract-matching can be simply represented as language inclusion. Given two automata Aut^C and Aut^P representing respectively a formal rule specification of a contract (Spec^C) and of a policy (Spec^P), $\text{Spec}^C \sqsubseteq \text{Spec}^P$ when $\mathcal{L}_{\text{Aut}^C} \subseteq \mathcal{L}_{\text{Aut}^P}$, i.e. the language accepted by Aut^C is a subset of the language accepted by Aut^P. Informally, each behavior of Aut^C is among the behaviors that are allowed by the policy Aut^P. Assuming that the automata are closed under intersection and complementation, then the matching problem can be reduced to an emptiness test [8]:

$$\mathcal{L}_{\text{Aut}^C} \subseteq \mathcal{L}_{\text{Aut}^P} \Leftrightarrow \mathcal{L}_{\text{Aut}^C} \cap \overline{\mathcal{L}_{\text{Aut}^P}} = \emptyset \Leftrightarrow \mathcal{L}_{\text{Aut}^C} \cap \mathcal{L}_{\overline{\text{Aut}^P}} = \emptyset$$

In other words, there is no behavior of Aut^C that is disallowed by Aut^P. If the intersection is not empty, any behavior in it corresponds to a counterexample.



(a) Contract in Example 2.2 (b) Policy in Example 2.2

Fig. 7. Automata specifying the rules of the contract and the policy of Example 2.2.

Our goal is to provide application-contract vs. platform-policy matching on-the-fly, namely during the actual download of the midlet. Thus, issues like small memory footprint, and effective computations play a key role. Therefore, we are interested in finding counterexamples faster and we combine algorithm based on Nested DFS [43] with decision procedure (for details see [33]). The algorithm takes as input the application's contract and the mobile platform's policy as automata and then starts a depth first search procedure over the initial state. When a "suspect state" (which is an accepting state) is reached we have two cases. First, when a suspect state contains an error state of the complemented policy ($\overline{\text{Aut}^P}$) then we report a security policy violation without further ado. Second, when a suspect state does not contain an error state of the complemented policy we start a new depth first search from the suspect state to determine whether it is in a cycle, i.e. it is reachable from itself. If it is we report availability violation.

The matching algorithm does not construct the product automaton explicitly but on-the-fly. The on-the-fly emptiness test (constructing product automaton while searching the automata) is lifted from the algorithm by Coucubertis et al. [6] with modification of this algorithm from Holzmann et al.'s [26] considered as state-of-the-art (used in Spin [25]). The complexity results if the theory $\mathcal{T}$ is decidable with an oracle for the decision problem in the complexity class $\mathcal{C}$ [33]:

- The non-emptiness problem is decidable in $LIN - TIME^{\mathcal{C}}$.
- The non-emptiness problem is $NLOG - SPACE^{\mathcal{C}}$.

An alternative approach would be to use simulations between the contract automata and the policy automata. Several notions of simulation relations for automata have been introduced in the literature ([15,14,10] to mention only a few) and discussing each of them is outside the scope of the paper. Intuitively, we can say that a state q_i of an automata A "simulates" a state q_j of an automata B if every "behavior" starting at q can be mimicked, step by step, starting at q_j.

The main approach for determining simulation relations among automata consists of reducing the simulation problem to a simulation game, i.e. to the problem of searching the winning strategy of a parity game graph [14]. We have adapted a simulation construct following the approach [15]. Basically, a parity game graph is constructed starting from two automata A and B and according to a well specific notion of simulation relation. Then the Jurdzinski Algorithm [28] is used for determining the set of winning nodes.

Example 6.2. Let us focus on Example 2.2. The ConSpec specifications of the contract and the policy (Fig. 4a and b, respectively) are translated into the two automata represented in Fig. 7. If we use language inclusion for matching then we have to complement the policy automaton on Fig. 7b, where we have only one accepting state, namely error and the complementation transitions remain the same as the original transitions. Afterwards, we check the emptiness of intersection between automaton on Fig. 7a and the complemented policy automaton. Since the emptiness check fails, then matching fails. Another means to check for matching is by simulation, where given the two automata we construct a game graph and search for a winning strategy for a protagonist. If automaton of policy (Fig. 7b) can simulate automaton of contract (Fig. 7a) then matching succeeds, where in our current example the simulation fails.

7. Software architecture

In this section we describe the software architecture of the prototype that implements the overall matching algorithm. The main aim is to provide a concrete and high-level overview of how the prototype works, from its inputs (ConSpec specifications)

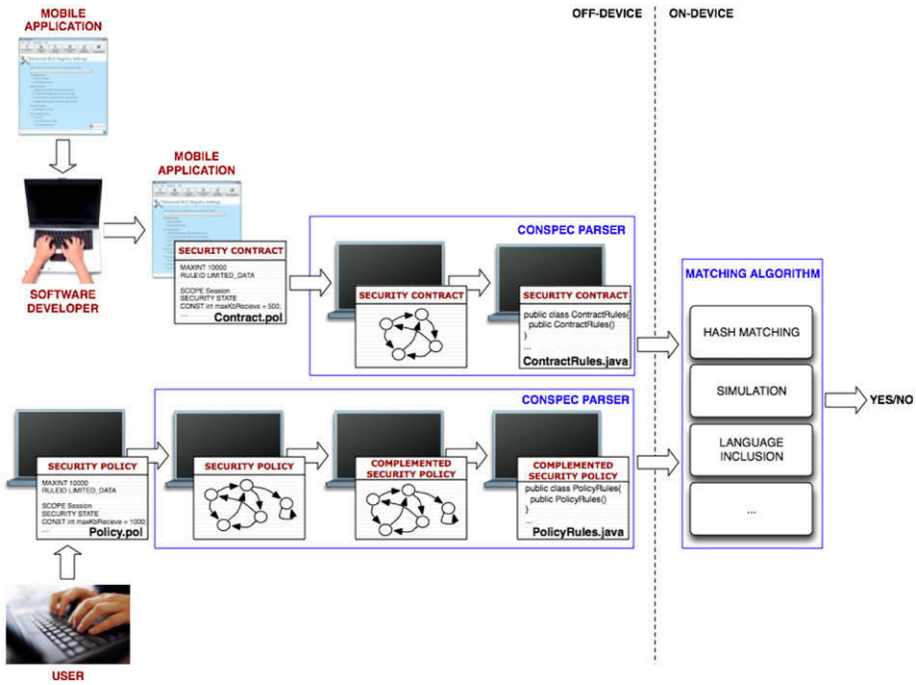


Fig. 8. Architecture of the matching prototype.

to its output (match succeeded/failed). To this purpose, we will also show what happens when the prototype is executed with our running examples as inputs.

As remarked in Section 4, there can be many ways to do the actual formal matching (simulation, language inclusion, ...). Although the prototype has been implemented with several matching algorithms, for the sake of simplicity from now on we will focus only on matching as language inclusion (as discussed in Section 6).

The contract-matching prototype takes as input a contract and a policy both specified in ConSpec and checks whether or not the contract matches the policy. A sketch of the prototype architecture is shown in Fig. 8.

As the computational resources of mobile devices are normally limited and performing heavy computations exhausts the battery, the motivation behind the design of our prototype has been to move as much operations as possible off the device. For this reason, the prototype is basically composed of two tools: an off-device ConSpec parser and the main matching algorithm that runs on the device. The parser takes as input a ConSpec file and returns a Java source code file representing the automata-based rule set (that is, the list of rules grouped by scope of the contract/policy, where each rule is specified by means of an automaton).

The matching algorithm needs as inputs (1) a rule of the contract and (2) a rule of the policy. Since we are using the language inclusion method, the rule representing the policy needs to be complemented. The complementation of the policy needs to be done only once at the time of deployment of the policy on the mobile device and therefore is performed directly by the off-device parser. This can be done by simply calling the parser with a specific parameter 'c' or 'p' (along with the path to the ConSpec file) that indicates whether it is a contract or a policy in input. The parsing procedure creates two Policy class instances, each of them contains the list of rules grouped by scope. Each rule instance contains the corresponding automaton (AutomatonMTT class instance) that is needed for the inclusion.

When the main algorithm starts, the contract and the policy represented as Policy class instances are already part of the program codebase and the matching procedure can begin. No computational resources are spent on parsing the specification or complementing the policy automaton. For every rule in the policy we must have a corresponding rule in the contract with same RULEID and SCOPE tags: if this is not the case the whole match fails, otherwise we perform the inclusion match on this pair.

If the inclusion match fails, the whole procedure halts with a "failure". If it succeeds the procedure continues with the next pair of rules. The described step is repeated until all rules in the policy have been successfully checked against the rules in the contract.

Example 7.1. Back to our running examples, let us consider Example 2.1. Given a contract and a policy in ConSpec (Fig. 3a and b) as inputs, the prototype translates these specifications into automata (Fig. 6). Such automata are then used as inputs of the language inclusion algorithm. Specifically, for each rule in the policy we search for corresponding rule in the contract and run

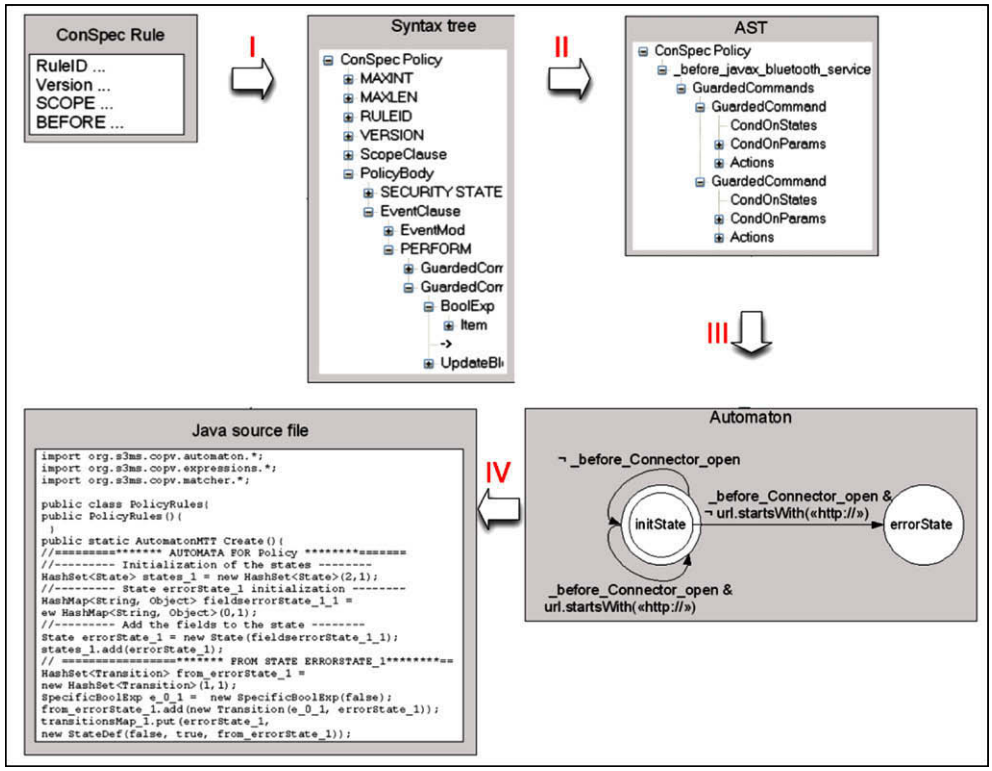


Fig. 9. ConSpec parser.

emptiness checking algorithm on the corresponding two rules. The first pair of rules with the same `RuleID` is the one showed in Fig. 3a and b. Since the contract allows to use HTTPS connections only while the policy allows to use both HTTP and HTTPS connections the obtained result states that the contract matches the policy. For the other rule in the policy an appropriate rule in the contract is found. Here the contract forbids the application to send messages while the policy prescribes that the application can send bounded amount of messages. As a result, the matching algorithm ends successfully: the contract matches the policy.

Example 7.2. Let us focus on Example 2.2. The ConSpec specifications of Fig. 4a and b are translated into the two automata represented in Fig. 7. Here the matching fails because the algorithm finds a cycle. This is because the contract allows to receive more data then the policy.

The parser implements the mapping from a ConSpec contract/policy to a Java source file (list of AutomatonMTT class instances). A sketch of the implemented parser is shown in Fig. 9. At first step (I), a syntax tree containing all the significant items of the contract/policy is made.

In the second transformation (II), the parser finds all the events and builds a specific structure called AST. Each event now has a list of guarded commands. Each guarded command consists of condition on state variables, condition on parameters of the method and actions for the guard.

During the next step (III), the automaton is built from the AST. First, we generate the list of expressions that will be used for creating the transitions taking into account that only one security event at a time may happen. Second, we create all the states and then all the transitions for every pair of state and generated expression. The detailed mapping procedure is outside the scope of the paper. Interested readers can find it in [12].

Finally, the last step (IV) creates the Java source code containing the instance of the automaton.

The language inclusion matching interacts with the SMT solver NuSMV[6] for satisfiability checks as shown in Algorithms 5 and 6. We create the instance of the NuSMV class only once at the beginning of the on-the-fly procedure; then we declare variables, add and remove constraints from the library every time we call the solver. Constraints for the solver are often repeated during the running algorithm (at least the constraints for searching for accepting state and searching for cycles are the same). To avoid calling the solver frequently for the same problem we added two lists in the DFSAlgorithm class:

[6] http://nusmv.irst.itc.it/.

Algorithm 5 DecisionProcedure$(a1, a2)$ Procedure

Input: SpecificBoolExp $a1$, SpecificBoolExp $a2$;

```
1: String s := a1.toString() + "&" + a2.toString();
2: if (Table_SAT.contains(s)) then
3:    return true;
4: else
5:    if (Table_UNSAT.contains(s)) then
6:       return false;
7:    else
8:       return CallNuSMV(a1, a2, s);
9:    end if
10: end if
```

Algorithm 6 CallNuSMV$(a1, a2, s)$ Procedure

Input: SpecificBoolExp $a1$, SpecificBoolExp $a2$, String s;

```
1: a1.DeclareVariables(nuSMV);
2: a2.DeclareVariables(nuSMV);
3: nuSMV.AddConstraint(s);
4: if (nuSMV.Solve()) then
5:    Table_SAT.add(s);
6:    return true;
7: else
8:    Table_UNSAT.add(s);
9:    return false;
10: end if
```

Table_SAT and Table_UNSAT. The Table_SAT contains the constraints that are checked by the solver and result is SAT. Similarly, Table_UNSAT contains the constraints that are checked by the solver and result is UNSAT.

We faced a number of design options:

`ONE_INSTANCE versus MANY_INSTANCES`**.** We could either create only one instance of solver, relying on the solver to assert and retract expressions on demand, or create a new instance of the solver every time we call the decision procedure.

`MUTEX_SOLVER`**.** Method names are declared as mutex constants at the moment of declaring all variables on the solver, due to an edge in the automaton (correspond to a method) which is incompatible with another edge (correspond to a different method).

`MUTEX_MC`**.** Method names are not declared as mutex constants, allowing the on-the-fly algorithm to check whether method names are the same.

`PRIORITY_MC`**.** Guards are evaluated using *priority or* and we can optimize the expressions sent to the decision procedure as minimized expression.

`CACHING_MC`**.** We saved time by caching the results of the matching.

`CACHING_SOLVER`**.** The solver itself has a caching mechanism that could be equally used.

However, `MANY_INSTANCES` decision was not possible to be taken because of the garbage collection management both by the Java virtual machine and by the libraries of MathSAT/NuSMV (only one instance of solver exists at time).

We ran our experiments on a Desktop PC (Intel(R) Pentium(R) D CPU 3.40 GHz, 3389.442 MHz, 1.99 GB of RAM, 2048 KB cache size) with operating system Linux version 2.6.20-16-generic, Kubuntu 7.04 (Feisty Fawn). The desktop implementation was mostly used for profiling and extracting details on the algorithm as a profiling tool was not available on the mobile device.

We also ported the prototype to the mobile, namely HTC P3600 (3G PDA phone) with ROM 128 MB, RAM 64 MB, Samsung®SC32442A processor 400 MHz and operating system Microsoft Windows Mobile®5.0.

The experiments were made for different design decisions for different problem suites (see [4] for details). The problem suites covered different kind of categories such as network connectivity, use of costly functionalities, or private information management.

We collected data on resources used, namely number of visited states, number of visited transitions, running time for each problem in each design alternative, and the number of solved problems against time. We present the result obtained for alternative with `MUTEX_MC`, `ONE_INSTANCE` and `CACHING_SOLVER` in Table 1. Most problems have few states and transitions as a result the matching algorithm runtime is little (around 2.5 s on the desktop and 4 s on the mobile). Even for pathological problem with 102 visited states, the runtime on a mobile platform is still acceptable (9.3 s on the desktop, 11.3 s on the mobile).

Table 1
Running problem suite 10 times.
(a) Running Problem Suite.

Problem	Desktop				Mobile				Result
	ART (s)	CRT (s)	SV	TV	ART (s)	CRT (s)	SV	TV	
MUTEX_MC ONE_INSTANCE CACHING_SOLVER									
P1	2.4	2.4	2	6	4.3	4.3	2	6	Match
P2	2.4	4.8	2	6	4.1	8.4	2	6	Match
P3	2.4	7.2	3	11	3.9	12.3	3	11	Match
P4	2.4	9.6	2	6	4.0	16.3	2	6	Match
P5	4.7	14.3	3	11	4.1	20.4	3	11	Match
P6	2.9	2.9	4	4	3.8	3.8	3	6	Not match
P7	2.8	5.7	5	7	3.8	7.6	2	4	Not match
P8	2.9	8.6	5	7	3.8	11.4	3	6	Not match
P100	9.3	9.3	102	307	11.3	11.3	102	307	Match

(b) Abbreviations.
ART: Average Runtime for 10 runs SV: Number of Visited States.
CRT: Cumulative Average Runtime TV: Number of Visited Transitions.

While the timing of the desktop is obviously faster, the cumulative runtime of solved problems is still manageable for the mobile user. An expected nice feature confirmed by the experiments is that the runtime will be longer for the problems that match (the algorithm has to run over all states until the cycle is found) than for the problems that do not match (the algorithm stops working as soon as counterexample is found). In the matching case the user has to wait longer but at least those are the applications that he would later run.

8. Related work and conclusions

Contracts are formal agreements to express the relation between a supplier and its clients, expressing each party's rights and obligations. Following the classification proposed in [5] there are basic or syntactic contracts, assertion-based contracts, behavioral contracts, synchronization contracts and quality-of-service (QoS) contracts.

Syntactic contracts are required for basic component interoperability. Contracts of this level specify the types, structures and signatures of components and methods through *interfaces* and *types definitions*.

Assertion-based contracts are the original contracts proposed by Bertrand Meyer [7,35]. A contract between a routine and its callers is defined by two assertions: a precondition and a postcondition. In addition to contracts bound to a single method, global properties on each instance of a class can be expressed using an assertion named class invariant.

Behavioral contracts [22] add the specification of the entities involved and an ordered sequence of the message exchanged by those entities.

Synchronization contracts [16] describe concurrent relations between services provided by components (sequence, parallelism or shuffle). They can be bound to a single method, declaring under which circumstances the method accesses a resource or they can be bound to a collection of methods or objects.

QoS contracts can be specified statically by enumerating the features that the server objects will respect or dynamically by negotiation between the object client and its server [5].

The notion of contract proposed in this paper is essentially those of behavioral contracts.

In the realm of security research, five main approaches to mobile code security can be broadly identified in the literature: *sandboxes* limit the instructions available for use, *code signing* ensures that code originates from a trusted source, *security automata* proscribes execution of mobile code containing violations of the security policy, *proof-carrying code (PCC)* carries explicit proof of its safety and *model-carrying code (MCC)* carries security-relevant behavior of the mobile code.

The *Sandbox Security Model* is the original security model provided by Java. The essence of the approach [17] is that a computer entrusts local code with full access to vital system resources (such as the file system). It does not, however, trust downloaded remote code (such as applets), which can access only the limited resources provided inside the sandbox. The limitation of this approach is that it can provide security but only at the cost of unduly restricting the functionality of mobile code (e.g. the code is not permitted to access any files). The sandbox model has been subsequently extended in Java 2 [18], where permissions available for programs from a code source are specified through a security policy. Policies are decided solely by the code consumer without any involvement of the producer. The implementation of security checking is done by means of a run-time *stack inspection* technique [47].

In .NET [30] each assembly is associated with some default set of permissions according to the level of trust. However, the application can request additional permissions. These requests are stored in the application's *manifest* and are used at load-time as the input to policy, which decides whether they should be granted. Permissions can also be requested at run-

time. Then, if granted, they are valid within the limit of the same method, in which they were requested. The set of possible permissions includes, for instance, permissions to use sockets, web, file IO, etc.

Cryptographic code-signing is the complement of sandboxing. It is widely used for certifying the origin (i.e., the producer) of mobile code and its integrity. Typically, the software developer uses a private key to sign executable content. The application loading the module then verifies this content using the corresponding public key and then the code can run with the full privileges associated with the signature.

This technique is useful only for verifying that the code originated from a trusted producer and it does not address the fundamental risk inherent to mobile code, which relates to mobile code behavior. This leaves the consumer vulnerable to damage due to malicious code (if the producer cannot be trusted) or faulty code (if the producer can be trusted). Indeed, if the code originated from an untrusted or unknown producer, then code-signing provides no support for safe execution of such code. On the other hand, code signing does not protect against bugs already present in the signed code. Patches or new versions of the code can be issued, but the loader (which verifies and loads the executable content and then transfers the execution control to the module) will still accept the old version, unless the newer version is installed over it. [36] proposes a method that employs an executable content loader and a short-lived configuration management file to address this software aging problem.

Another mechanism for security property enforcement is security automata [42,19]. The automaton itself is a finite state automaton, whose input is an event stream of security-relevant actions. Now the mobile code is not accompanied by additional information, it is only executed in tandem with simulation of the security automaton. If action stream in the given code violates automaton then the execution of untrusted code terminates. A wider class of automata were presented to enable modifications of code execution: *edit automata* [3] may terminate the application, truncating the program action stream (as a simple security automata); suppress undesired actions even without termination of the program; and they may insert additional actions into the event stream. Those enforcement mechanisms complement the matching aspect that we describe here and are essentially used for the run-time monitoring step that we have presented in Fig. 1.

It is not necessary to use finite state automata for run-time monitoring. For instance, temporal logic formulae are widely applied for this purpose [21]. Since there is a mapping from temporal logic to FSA, one can translate policies written as logic formulae into automata-based language and vice versa.

The Proof Carrying Code (PCC) approach [39] launched the idea that untrusted code is accompanied by additional information that aids in verifying its safety. Within PCC, this additional information takes the form of a proof regarding the safety of mobile code. Then the code consumer uses a proof validator to check, with certainty, that the proof is valid (i.e., it checks the correctness of this proof) and hence the foreign code is safe to execute. Proofs are automatically generated by a certifying compiler [40] by means of a static analysis of the producer code. The traditional approach to PCC based on type theory is problematic for two main reasons noted by Sekar et al. [44]. A practical difficulty is that automatic proof generation for complex properties is still a daunting problem, requiring expertise in logic and theorem proving that is beyond reach for normal developers. A more fundamental difficulty is that the type-theory approach is based on an unrealistic assumption [44]: the safety proof is based on a type system and this type system cannot be configured for each individual policy. This appears an impractical assumption since security may vary considerably across different consumers and their operating environments.

Model Carrying Code (MCC) is the other seminal work beside Meyer's one on which our proposal is based that requires a producer to furnish a model regarding the safety of mobile code [45]. With MCC, this additional information captures the *security-relevant behavior* of the code. Models enable code producers to communicate the security needs of their code to the consumer. The code consumers can then check their policies against the model associated with untrusted code to determine if this code will violate their policy. Since MCC models are significantly simpler than programs, such checking can be fully automated. This model has been mainly proposed for bridging the gap between high-level policies and low-level binary code, enabling analyses which would otherwise be impractical. The major limitation was that MCC had not fully developed the issue of contract matching and had limited itself to finite state automata which are too simple to describe realistic policies.

The notion of certified inline monitors [48,19] can bridge the gap between security-by-contract and proof-carrying-code. Essentially, an inline monitor rewrites the application code by inserting at specific points the security checks [13]. Such inlining can generate in parallel a proof that the code complies with the contract that it is actually inlined. Then one can ship the proof of the compliance of the contract with the application and the contract. Proof-checking techniques can then be used to check the evidence in the corresponding step of Fig. 1. This has been currently implemented in [9].

A similar technique to inline monitoring is used in Aspect Oriented Programming (e.g. AspectJ) and in the implementation of programming-by-contract in Java [1]. Loosely speaking the API calls that an inlined security automata would like to monitor can be compared to point-cuts in aspects.

In this paper we have proposed the notion of *security-by-contract* where an application carries a mobile contract with itself. The key idea of the approach is that a digital signature should not only certify the origin of the code but rather bind the code with the contract. From this point of view, our framework makes some ideas behind MCC more concrete. In particular, we use a high level specification language with features that simplify contract-policy matching and allow expressing realistic security policies. In the original work on MCC, the models that represent the behavior of the application are represented as extended finite state automata (EFSA). However, the developed matching algorithms for EFSA handle only simple cases when the conditions on transactions are limited to equalities/inequalities [45]. In contrast, our approach allows us to express much more sophisticated conditions that include basic arithmetic and string operations. Furthermore, our matching algorithm is

improved for efficiency intended for resource-critical devices as mobiles. Thus, we first perform easier checks of sufficient criteria before a complete check (see Algorithm 4). These features differentiate our approach from other frameworks for modeling resource contractualization such as [31].

The contributions of the paper are manifold. First, we have proposed the *security-by-contract* framework providing a description of the overall life-cycle of mobile code in this setting. Then we have described a structure for a contractual language. Starting from this language, we have proposed a number of algorithms for one of the key steps in the life-cycle process: the issue of *contract-policy matching*. Finally, we have discussed the software architecture supporting all these algorithms and evaluated its desktop and mobile implementation.

The paper can be seen as an updated summary of several papers describing different S×C technologies [2,4,11,33,34]. For instance, the algorithms presented here as well as their implementation have been improved and optimized. Therefore, the paper represents a first, complete and updated presentation of the overall S×C contract-policy matching technology.

The main novelty of the proposed framework is that it would provide a semantics for digital signatures on mobile code thus being a step in the transition from trusted code to trustworthy code.

Acknowledgement

The authors would like to thank Marco Dalla Torre (University of Trento) for his contribution on the development of the first prototype of the software architecture.

References

[1] P. Abercrombie, M. Karaorman, Jcontractor: bytecode instrumentation techniques for implementing design by contract in java, in: Proceedings of Second Workshop on Runtime Verification, ENTCS, 2002, pp. 55–79.
[2] I. Aktug, K. Naliuka, ConSpec – a formal language for policyspecification, in: Proceedings of the 1st Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007) , ENTCS, 2008, pp. 45–58.
[3] L. Bauer, J. Ligatti, D. Walker, Edit automata: enforcement mechanisms for run-time security policies, Internat. J. Inform. Security, 4 (1–2) (2005) 2–16.
[4] N. Bielova, I. Siahaan, Testing decision procedures for security-by-contract, FCS-ARSPA-WITS'08, 2008.
[5] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, D. Watkins, Making components contract aware, Computer 32 (7) (1999) 38–45.
[6] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal Methods System Des. 1 (2–3) (1992) 275–288.
[7] B. Meyer, Building bug-free O–O software: an introduction to design by contract. Available at: http://archive.eiffel.com/doc/manuals/technology/contract/.
[8] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 2000.
[9] M. Dam, A. Lundblad, A proof carrying code framework for inlined reference monitors in Java bytecode, Technical Report, KTH, 2008.
[10] D.L. Dill, A.J. Hu, H. Wong-Toi, Checking for language inclusion using simulation relations, in: Proceedings of CAV'91: 3rd International Workshop on Computer Aided Verification, LNCS, 1992, pp. 329–341.
[11] N. Dragoni, F. Massacci, K. Naliuka, I. Siahaan, Security-by-contract: toward a semantics for digital signatures on mobile code, in: Proceedings of the 4th European PKI Worshop, LNCS, 2007, pp. 297–312.
[12] N. Dragoni, F. Massacci, K. Naliuka, I. Siahaan, T. Quillinan, I. Matteucci, C. Schaefer, Methodologies and tools for contract matching, Public Deliverable D2.1.4, EU Project S3MS, 2007. Report available at: www.s3ms.org.
[13] U. Erlingsson, The inlined reference monitor approach to security policy enforcement, Technical Report 2003-1916, Department of Computer Science, Cornell University, 2003.
[14] K. Etessami, A hierarchy of polynomial-time computable simulations for automata, in: Proceedings of CONCUR'02, LNCS, 2002, pp. 131–144.
[15] K. Etessami, T. Wilke, R. Schuller, Fair simulation relations, parity games, and state space reduction for buchi automata, in: Proceedings of ICALP'01, LNCS, 2001, pp. 694–707.
[16] H. Giese, J. Graf, G. Wirtz, Contract-based coordination of distributed object systems, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), CSREA Press, 1999, pp. 25–31.
[17] L. Gong, Java security: present and near future, IEEE Micro 17 (3) (1997) 14–19.
[18] L. Gong, G. Ellison, Inside Java 2 Platform Security: Architecture, API Design, and Implementation, Pearson Education, 2003.
[19] K. Hamlen, G. Morrisett, F. Schneider, Certified in-lined reference monitoring on .NET, in: PLAS '06: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, ACM Press, 2006, pp. 7–16.
[20] K.W. Hamlen, G. Morrisett, F.B. Schneider, Computability classes for enforcement mechanisms, ACM Trans. Program. Lang. Systems 28 (1) (2006) 175–205.
[21] K. Havelund, G. Rosu, Efficient monitoring of safety properties, Internat. J. Software Tools Technol. Transfer 6 (2) (2004) 158–173.
[22] R. Helm, I.M. Holland, D. Gangopadhyay, Contracts: specifying behavioral compositions in object-oriented systems, in: Proceedings of OOPSLA'90, ACM Press, 1990, pp. 169–180.
[23] T. Henzinger, O. Kupferman, S. Rajamani, Fair simulation, in: Proceedings of CONCUR'97, Academic Press, Inc., 1997, pp. 273–287.
[24] M. Hilty, A. Pretschner, C. Schaefer, T. Walter, Usage control requirements in mobile and ubiquitous computing applications, in: Proceedings of the International Conference on Systems and Net. Comm. (ICSNC 2006), IEEE Press, 2006, p. 27.
[25] G. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2004.
[26] G.J. Holzmann, D. Peled, M. Yannakakis, On nested depth first search, in: Proceedings of the Second SPIN Workshop, American Mathematical Society, 1996, pp. 23–32.
[27] T. Jensen, D.L. Metayer, T. Thorn, Verification of control flow based security properties, in: Proceedings of IEEE Symposium on Security and Privacy, IEEE, 1999, pp. 89–103.
[28] M. Jurdzinski, Small progress measures for solving parity games, in: Proceedings of STACS'00, LNCS, 2000, pp. 290–301.
[29] Y. Kesten, Z. Manna, H. McGuire, A. Pnueli, A decision algorithm for full propositional temporal logic, in: Proceedings of CAV, LNCS, 1993, pp. 97–109.
[30] B. LaMacchia, S. Lange, NET Framework Security, Addison Wesley, 2002.
[31] N. Le Sommer, Towards dynamic resource contractualisation for software components, in: Proceedings of the International Working Conference on Component Deployment (CD 2004), LNCS, 2004, pp. 129–143.
[32] F. Martinelli, I. Matteucci, Through modeling to synthesis of security automata, in: Proceedings of the 2nd International Workshop on Security and Trust Management, LNCS, 2006, pp. 31–46.
[33] F. Massacci, I. Siahaan, Matching midlet's security claims with a platform security policyusing automata modulo theory, in: Proceedings of NordSec'07, 2007.

[34] F. Massacci, I. Siahaan, Simulating midlet's security claims with automata modulo theory, Proceedings of the Workshop on Programming Languages and Analysis for Security, ACM Press, 2008, pp. 1–9.
[35] B. Meyer, Object Oriented Software Construction, second ed., Prentice Hall, 1997.
[36] J.R. Michener, T. Acar, Managing System and Active-Content Integrity, IEEE Comput. 33 (7) (2000) 108–110.
[37] MOBIUS Project Team, Framework- and application-specific securityrequirements, Public Deliverable of EU Research Project D1.2, MOBIUS – Mobility, Ubiquity and Security, 2006. Report available at: http://mobius.inria.fr.
[38] T. Moses, eXtensible Access Control Markup Language (XACML) version 1.0, Technical report, OASIS, 2003.
[39] G.C. Necula, Proof-carrying code, in: Proceedings of POPL '97, ACM Press, 1997, pp. 106–119.
[40] G.C. Necula, P. Lee, The design and implementation of a certifying compiler, SIGPLAN 39 (4) (2004) 612–625.
[41] Bill Ray, Symbian signing is no protection from spyware, The Register, May 2007. Available on the web at: http://www.theregister.co.uk/2007/05/23/symbian_signed_spyware/.
[42] F. Schneider, Enforceable security policies, ACM Trans. Inform. System Security 3 (1) (2000) 30–50.
[43] S. Schwoon, J. Esparza, A note on on-the-fly verification algorithms, Technical Report 2004/06, University of Stuttgart, November 2004.
[44] R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Model-carrying code (MCC): a new paradigm for mobile-code security, in: NSPW '01: Proceedings of the 2001 Workshop on New Security Paradigms, ACM Press, 2001, pp. 23–30.
[45] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, Daniel C. DuVarney, Model-carrying code: a practical approach for safe execution of untrusted applications, ACM SIGOPS Oper. Syst. Rev. 37 (5) (2003) 15–28.
[46] Jan Smans, Bart Jacobs, Frank Piessens, Static verification of code access security policy compliance of .NET applications, J. Object Technol. 5 (3) (2006) 35–58.
[47] D.S. Wallach, E.W. Felten, Understanding Java stack inspection, in: Proceedings of IEEE Symposium on Security and Privacy, IEEE, 1998, pp. 52–63.
[48] D. Vanoverberghe, F. Piessens, A caller-side inline referencemonitor for an object-oriented intermediate language, in: Proceedings of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08), LNCS, 2008, pp. 240–258.
[49] D. Vanoverberghe, P. Philippaerts, L. Desmet, W. Joosen, F. Piessens, K. Naliuka, F. Massacci, A flexible security architecture to support third-party applications on mobile devices, Proceedings of the 1st ACM Computer Security Architecture Workshop, ACM Press, 2007, pp. 19–28.
[50] V.N. Venkatakrishnan, R. Peri, R. Sekar, Empowering mobile code using expressive security policies, Proceedings of the New Security Paradigms Workshop, ACM Press, 2002, pp. 61–68.
[51] B.S. Yee, A sanctuary for mobile agents, in: J. Vitek, C.D. Jensen (Eds.), Secure Internet Programming, LNCS, 2001, pp. 261–273.
[52] A. Zobel, C. Simoni, D. Piazza, X. Nunez, D. Rodriguez, Business case and security requirements, Public Deliverable D5.1.1, EU Project S3MS, October 2006. Report available at: www.s3ms.org.