Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code*

N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan

Department of Information and Communication Technologies
University of Trento, ITALY
surname@dit.unitn.it

Abstract. In this paper 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 overall life-cycle of mobile code in the setting of security-by-contract, describe a tentative 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 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 approach and is based on *trust relationships*: mobile code is run if its origin is trusted. This essentially boils down to *mobile code 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. E.g. 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*. 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. One might well trust

* Research partly supported by the project EU-IST-STREP-S3MS.

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, by signing the code they essentially declare that they did it. The consequence is that injecting an application in the mobile market is a time consuming operation as SME developers must essentially convince the operators that their code will not do anything harmful.

1.1 Contribution of the Paper

We propose in this paper the notion of *security-by-contract* (as in programming-by contract [1]): 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*[1], 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 tentative 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.

The research is performed within the limits of the European project "Security of Software and Services for Mobile Systems" (S3MS)[2].

The rest of the paper is organized as follows. In Section 2 we present the security-by-contract framework providing a description of the overall life-cycle of mobile code in this setting. In Section 3 we describe typical security requirements to mobile applications. In Section 4 we focus on contract specification defining also the notion of contract-policy matching. Then in Section 5 we propose an algorithm for contract-policy matching based on the contractual language of Section 4. We end the paper discussing related work and conclusions.

2 The Security-by-Contract Life-Cycle

The framework of security-by-contract for mobile code is essentially shaped by three groups of stake-holders: mobile operator, service provider and/or developer, mobile user. This is shown in Fig. 1.

[1] 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.

[2] More information concerning this project can be acquired at http:\\www.s3ms.org.

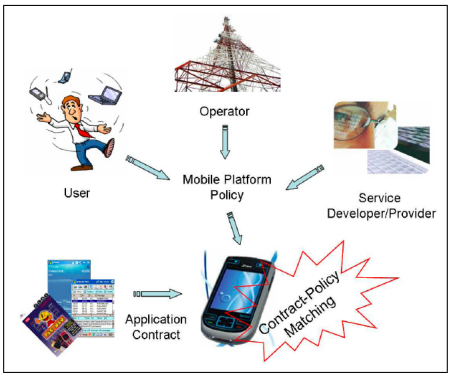

Fig. 1. Key Stakeholders

Table 1. Enforcing Security-by-Contract at Different Stages

Development	Deployment		Execution
(I) at design and development time	(II) after design but before shipping the application	(III) when downloading the application	(IV) during the execution of the application

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

Definition 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).*

By signing the code 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 that all codes that are deployed on their platform are secure. In other words they must declare their security policy:

Definition 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).*

A contract should be negotiated and enforced during development, at time of delivery and loading, and during execution of the application by the mobile platform. Fig. 2 summarizes the phases of the application/service life-cycle in which the contract-based security paradigm is present.

In order to guarantee that an application complies with its desired contract or the policy requested on a particular platform we should consider the stage where such enforcement can be done as shown in Table 1.

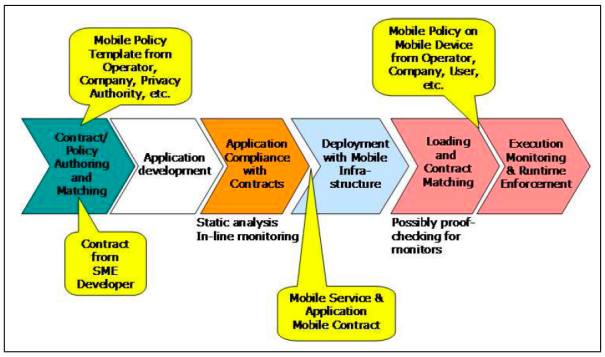

Fig. 2. Application/Service Life-Cycle

Enforcing at level (I) can be achieved by appropriate design rules and require developer support; (II) and (III) can be carried out through (automatic) verification techniques. Such verifications can take place before downloading (*static verification* by developers and operators followed by a contract coming with a *trusted signature*) or as a combination of pre and post-loading operations (e.g., through *in-line monitors* and *proof carrying code*); (IV) can be implemented by *run-time checking*. All methods have different technical and business properties. From an operator's view point:

- working on existing devices would rule out run-time enforcement and favour static analysis, code signing and signature verification on the mobile platform. Monitors may be used (for properties that could not be proved), but on-device proof would then not be possible.
- Operators distrusting the certification process could rely on run-time checks, at the price of upgrading devices' software. Monitors could be used and contracts could be verified on the device itself.
- An operator who wants to be able to run existing applications would prefer run-time enforcement.

The users' perspectives could be different as individuals might care more of privacy, whereas companies might care more of security. Their interest could be on ease of matching the mobile contract against the mobile policy or the combination of policies (e.g. operator, company, user or roaming on another operator's network). The Table 2 shows some of possible strengths and limitations of each different technology.

Contract-Policy Matching. As we can see in Fig. 2, one of the key problems in the overall security-by-contract life-cycle is the *contract-policy matching* issue: given a contract that an application carries with itself and a policy that a platform specifies, is the contract compliant with the policy? 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 platforms policy. Contract-policy matching represents a common problem in the life-cycle because it must be done at all levels:

Table 2. Enforcement Technology Strengths and Weaknesses

Criteria	Static Analysis	Monitors	Runtime
Works with existing devices	√	?	×
Works with existing applications	?	×	√
Does not modify applications	√	×	√
Offline proof of correctness	√	√	×
Load-time proof of correctness	×	√	×
May depend on run-time data	×	√	√
Does not affect runtime performance	√	?	×

both for development and run-time operation. To address this issue we need efficient algorithms to match application contracts with device policies. This will be the target of Section 5.

3 How a Contract Should Look Like?

If 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 device a complex procedure is likely to defy the very spirit of our proposal.

However, a detailed case study [20] in the booming real of Mobile Games shows that detailed contracts are not really necessary. The characteristic feature of these applications is that they need wide access to connectivity to execute correctly. However, the user still wants to control that this connectivity is not abused or misused. Therefore the same permission can be granted or not granted depending, for instance, on previous actions of the applet or some conditions on application environment.

Let us make two examples. Personal information security can be ensured by the following policies:

Example 1. "*No external connections are allowed if the application has accessed the user's personal information*". In this example granting of the permission depends on the applets' previous actions.

Example 2. Using wireless connection can make the device runs out of battery very quickly. To prevent it one can apply the following policy: "*The application is not allowed to use wireless connection if the battery level is below a certain limit*". In this example the permission to run for the application depends on the state of its environment.

Other examples of security policies for mobile devices include:

1. The application sends no more than a number 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 name matches 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.

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 contracts 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*). So three possible scopes of the obligation are:

Multisession – the obligation must be fulfilled by all runs of the application as a whole.
Session – the obligation must be fulfilled by each run of the application separately.
Object – the obligation must be fulfilled by each object of a given type.

Another important issue is the *granularity* of the protected resources. The requirements above show that the field of application of the policy can be limited to particular services (HTTP, SMS, etc.) or even API calls (policy 8). Hence the fine-grained control over the protected resources is an important requirement to the security framework.

4 Contract Specification

A single contract/policy is specified as a *list of disjoint rules* (for instance rules for connections, rules for PIM and so on) instead of one giant specification describing all possible security properties. A rule is defined according to the following grammar:

```
<RULE> :=
    SCOPE [ OBJECT <class> |
        SESSION |
        MULTISESSION ]
        RULEID <identifier>
        <formal specification>
```

Rules can differ both by `SCOPE` and `RULEID`.Scope definition reflects at which scope (`OBJECT`, `SESSION`, `MULTISESSION`) the specified contract will be applied. The tag `RULEID` identifies the *area* of the contract (which security-relevant actions the policy concerns, for example “files” or “connections”).

We assume that `SCOPE` and `RULEID` divide the set of security-relevant actions into *non-interleaving sets* so that two rules with different scopes and `RULEID`s (in

the same contract specification) cannot specify the same security-relevant actions. This assumption allows us to perform matching as a number of simpler matching operations on separate rules, as we will show in Section 5.

For `SESSION` and `MULTISESSION` scopes there is a possibility to set `RULEID` equal to *, which means that the contract can include the actions from any area. Still we recommend to consider such an option carefully before using it because the matching of such contracts and policies can be inefficient.

The `<formal specification>` part of a rule gives a rigorous and not ambiguous definition of the behavior (semantics) of the rule. Since several semantics might be used for this purpose (such as standard process algebras, security automata, Petri Nets and so on), for the limited scope of this paper we abstract from a particular formal specification, identifying the necessary abstract constructs for combining and comparing rules. Moreover, we assume that rules can be combined and compared for matching only if they have the same scope. This assumption allows us to reduce the problem of combining rules to the one of combining their formal specifications, without considering scopes. Therefore the first thing we do when analyzing the specifications is to group rules within one scope together and reason about them separately.

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

- [Combine Operator $\oplus$] Spec $= \oplus_{i=1,\ldots,n}\text{Spec}_i$
  It combines all the rule formal 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 rule formal specification Spec^C simulates rule formal specification Spec^P, 0 otherwise.
- [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] $\mathcal{S} = \mathsf{Traces}\,(Spec)$
  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 1. $\mathsf{Traces}\,(\text{Spec}_1 \oplus \text{Spec}_2) = \mathsf{Traces}\,(\text{Spec}_1) \cup \mathsf{Traces}\,(\text{Spec}_2)$

Property 2. $\text{Spec}_1 \sqsubseteq \text{Spec}_2 \Leftrightarrow \mathsf{Traces}\,(\text{Spec}_1) \subseteq \mathsf{Traces}\,(\text{Spec}_2)$

Property 3. $\text{Spec}_1 \approx \text{Spec}_2 \Rightarrow \mathsf{Traces}\,(\text{Spec}_1) \subseteq \mathsf{Traces}\,(\text{Spec}_2)$

Definition 3 (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:*

$$\mathit{Traces}\left(\oplus_{i=1,\ldots,n} Spec_i^C\right) \subseteq \mathit{Traces}\left(\oplus_{i=1,\ldots,m} Spec_i^P\right)$$

Table 3. Examples of Contract/Policy Matching

Contract/Policy Rule	Object can use one type of connection only	Object can use every type of connections
Object can use HTTP connections only	☑	☑
Object can use HTTP and SMS connections	☐	☑

Definition 4 (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 satisfies the policy.*

Definition 5 (Complete Matching). *Matching should succeed if by executing the application on the platform every traces satisfying the contract also satisfy the policy.*

By applying Def. 4 we might reject "good" applications that are however too difficult or too complex to perform. On the other hand, Def. 5 may allow "bad" 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 are shown in Table 3.

5 Contract Matching Algorithm

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). In other words, the algorithm is defined by means of the abstract constructs discussed in Section 4. 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 5.1 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 automata.

As shown in Fig. 3, 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 represents 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: $\left\langle \mathcal{R}_{SESSION}, \mathcal{R}_{MULTISESSION}, \{\mathcal{R}_{class}\}_{class \in \zeta^C} \right\rangle$. As discussed in Section 4, this partition is necessary because in the S3MS framework comparison of rules starts only within a certain scope. 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

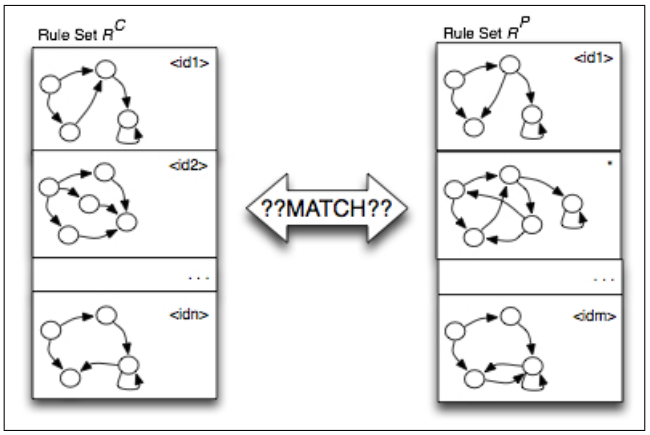

Fig. 3. Contract/Policy Matching Problem

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), than the contract matches the policy. Otherwise, matching fails.

Matching Rules with the Same Scope. 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 $\left(\text{ID}^{C/P},\ \text{Spec}^{C/P}\right)$ 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).

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.

Matching Specifications. The MatchSpec function (Algorithm 4) checks the match between a set of pairs $\mathcal{L}^C = \left\langle \left(\text{ID}^C_1,\ \text{Spec}^C_1\right), \ldots, \left(\text{ID}^C_n,\ \text{Spec}^C_n\right)\right\rangle$ and a pair $\left(\text{ID}^P,\ \text{Spec}^P\right)$ 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 $\left(\text{ID}^C,\ \text{Spec}^C\right)$ in L^C that matches with $\left(\text{ID}^P,\ \text{Spec}^P\right)$

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

$\left\langle \mathcal{R}^C_{SESSION}, \mathcal{R}^C_{MULTISESSION}, \{\mathcal{R}^C_{class}\}_{class \in \zeta^C} \right\rangle \Leftarrow \mathsf{Partition}(\mathcal{R}^C)$
$\left\langle \mathcal{R}^P_{SESSION}, \mathcal{R}^P_{MULTISESSION}, \{\mathcal{R}^P_{class}\}_{class \in \zeta^P} \right\rangle \Leftarrow \mathsf{Partition}(\mathcal{R}^P)$
if $\mathsf{MatchRules}(\mathcal{R}^C_{SESSION}, \mathcal{R}^P_{SESSION})$ **then**
  if $\mathsf{MatchRules}(\mathcal{R}^C_{MULTISESSION}, \mathcal{R}^P_{MULTISESSION})$ **then**
    for all $class \in \zeta^P$ **do** // *for all classes in policy*
      if $\mathsf{MatchRules}(\mathcal{R}^C_{class}, \mathcal{R}^P_{class})$ **then** // *if* $class \notin \zeta^C$, *then* $\mathcal{R}^C_{class} = \emptyset$
        skip
      else
        return(0)
      end if
    end for
    return(1)
  end if
end if
return(0)

Algorithm 2 Partition Procedure

Input: rule set $\mathcal{R}$
Output: $\left\langle \mathcal{R}_{SESSION}, \mathcal{R}_{MULTISESSION}, \{\mathcal{R}_{class}\}_{class \in \zeta} \right\rangle$

$\mathcal{R}_{SESSION} \Leftarrow \{r \in \mathcal{R} \mid \mathrm{Scope}(r) = \mathrm{SESSION}\}$
$\mathcal{R}_{MULTISESSION} \Leftarrow \{r \in \mathcal{R} \mid \mathrm{Scope}(r) = \mathrm{MULTISESSION}\}$
for all $class \in \zeta$ **do** // *for all classes in contract/policy*
  $\mathcal{R}_{class} \Leftarrow \{r \in \mathcal{R} \mid \mathrm{Scope}(r) = \mathrm{OBJECT} < class >\}$
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

$L^C \Leftarrow \{(\mathrm{ID}^C,\ \mathrm{Spec}^C) \mid \langle \mathrm{scope}, \mathrm{ID}^C, \mathrm{Spec}^C \rangle \in \mathcal{R}^C\}$
$L^P \Leftarrow \{(\mathrm{ID}^P,\ \mathrm{Spec}^P) \mid \langle \mathrm{scope}, \mathrm{ID}^P, \mathrm{Spec}^P \rangle \in \mathcal{R}^P\}$
for all $(\mathrm{ID}^P,\ \mathrm{Spec}^P) \in L^P$ **do**
  if $\mathsf{MatchSpec}(L^C, (\mathrm{ID}^P,\ \mathrm{Spec}^P))$ **then**
    skip
  else // *may return* $\emptyset$ *for efficiency*
    $L^P_{failed} \Leftarrow L^P_{failed} \cup (\mathrm{ID}^P,\ \mathrm{Spec}^P)$
  end if
end for
if $L^P_{failed} = \emptyset$ **then**
  return(1)
else
  return $\left(\mathsf{MatchSpec}\left(\left(*,\ \oplus_{(\mathrm{ID}^C,\ \mathrm{Spec}^C) \in L^C}\right), \left(*,\ \oplus_{(\mathrm{ID}^P,\ \mathrm{Spec}^P) \in L^P_{failed}}\right)\right)\right)$
end if

Algorithm 4 MatchSpec Function

Input: $L^C = \langle (\mathrm{ID}_1^C,\ \mathrm{Spec}_1^C), \ldots, (\mathrm{ID}_n^C,\ \mathrm{Spec}_n^C) \rangle$, $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$
Output: 1 if L^C matches $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$, 0 otherwise

```
1: if ∃ (ID^C, Spec^C) ∈ L^C ∧ ID^C = ID^P then
2:     if HASH(Spec^C) = HASH(Spec^P) then
3:         return(1)
4:     else if Spec^C ≈ Spec^P then
5:         return(1)
6:     else if Spec^C ⊑ 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( (∗, ⊕_{(ID^C, Spec^C) ∈ L^C}), (∗, Spec^P) )
13: end if
```

2. the combination of all the specifications in L^C matches with $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$

Otherwise, the function returns 0.

Specification matching is verified as follows. If there exists a pair $(\mathrm{ID}^C,\ \mathrm{Spec}^C)$ 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 $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$ (line 12).

5.1 Applying the Generic Matching Algorithm to Automata-based Rule Specifications

In this Section we show how the matching algorithm can be used when the behavior of rules (`<formal specification>`) is specified by means of finite state automata (FSA). In this way we provide a complete algorithm for matching contracts with FSA-based rule specifications. As already remarked at the beginning of Section 5, 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 FSA. Then we provide algorithms for implementing the abstract constructs.

FSA are widely used as a powerful formalism both for system modeling and for specification of system properties. Basically, FSA consists of finite numbers of states; transitions between states are performed through *actions*. A subset of states is selected

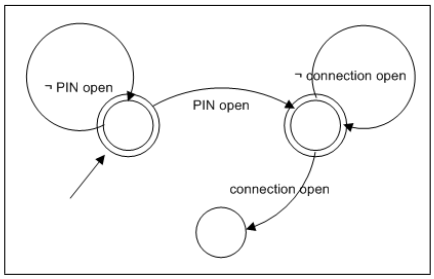

Fig. 4. The Automata-Based Specification of the Policy from Example 1 (circles with double borders denote accepting states, arrows represent the transition, where labels denote actions)

to be *accepting states*. If after performing a sequence of actions (a *run*) the FSA arrives in an accepting state then the automaton is said to *accept* this sequence of actions.

FSA that 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 language. For example, FSA for a temporal logic specification can be constructed using the *tableaux method* [11].

Example 3. To illustrate how FSA can be used for property specifying let us show the automaton for the policy from Example 1. The automaton has two accepting states. It remains in the first (initial) one until the action "PIN open" occurs. This action brings the automaton in the second accepting state. All actions in this state preserve it save the action "connection open". If "connection open" occurs the automaton is brought to the last (non-accepting) state, from which there is no outgoing transition. In this case the automaton terminates in a non-accepting states, which means that the automaton does not accept this run. The resulting automaton is presented at Fig. 4.

Combining Automata-Based Rules. 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.

Automata Matching as Language Inclusion. Given two automata Aut^C and Aut^P representing respectively a rule formal 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 [2]:

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

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. Algorithm 5 lists the basic steps for implementing the $\sqsubseteq$ operator for FSA-based rule specifications.

Algorithm 5 FSA-Based Implementation of $\sqsubseteq$

Input: two FSA-based rule specifications Aut^C and Aut^P
Output: 1 if $A_C \sqsubseteq A_P$, 0 otherwise
1: Complement the automaton Aut^P
2: Construct the automaton Aut^I that accepts $\mathcal{L}_{\text{Aut}^C} \cap \mathcal{L}_{\overline{\text{Aut}^P}}$
3: **if** Aut^I is empty **then**
4: return(1)
5: **else**
6: return(0)
7: **end if**

Algorithm 6 FSA-Based Implementation of $\approx$

Input: two FSA-based rule specifications Aut^C and Aut^P
Output: 1 if $A_C \approx A_P$, 0 otherwise.
1: Construct the automaton $\text{Aut}^U = \text{Aut}^C \cup \text{Aut}^P$
2: Build the parity game graph $G^*_A(q_0^{\text{Aut}^C}, q_0^{\text{Aut}^P})$
3: Compute the winning nodes W using Jurdzinski algorithm
4: **if** $(q_0^{\text{Aut}^C}, q_0^{\text{Aut}^P}) \in W$ **then**
5: return(1)
6: **else**
7: return(0)
8: **end if**

Automata Simulation. Several notions of simulation relations for automata have been introduced in literature ([3, 5, 4, 9] 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, i.e. $q_i \approx q_j \Rightarrow L(A[q_i]) \subseteq L(A[q_j])$.

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 nodes of a parity game graph [4]. Algorithm 6 summarizes the basic operations needed for implementing a simulation construct following this approach [5]. Basically, a parity game graph is constructed starting from two automata A and B and according to a well specific notion of simulation relation (the $*$ in the algorithm indicates a generic simulation relation). Then the Jurdzinski algorithm [10] is used for determining the set of winning nodes.

6 Related Work

Four main approaches to mobile code security can be broadly identified in literature: *sandboxes* limit the instructions available for use, *code signing* ensures that code originates from a trusted source, *proof-carrying code (PCC)* carries explicit proof of its safety, and *model-carrying code (MCC)* carries security-relevant behavior of the producer mobile code.

Sandbox Security Model. This is the original security model provided by Java. The essence of the approach [6] 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 [7], 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 [19].

In .NET 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. Cryptographic code-signing 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.

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. Patched 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. [12] proposes a method that employs an executable content loader and a short-lived configuration management file to address this software aging problem.

Proof-Carrying Code (PCC). The PCC approach [14] enables safe execution of code from untrusted sources by requiring a producer to furnish 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 [15] by means of a static analysis of the producer code. The PCC approach is problematic for two main reasons [16]. A practical difficulty is that automatic proof generation for complex properties is still a daunting problem, making the PCC approach not suitable for real mobile applications. A more fundamental difficulty is that the approach is based on a unrealistic assumption: since the producer sends the safety proof together with

the mobile code, the code producer should know all the security policies that are of interest to consumers. This appears an impractical assumption since security may vary considerably across different consumers and their operating environments.

Model-Carrying Code. This approach is strongly inspired by PCC, sharing with it the idea that untrusted code is accompanied by additional information that aids in verifying its safety [17]. With MCC, this additional information takes the form of a model that captures the *security-relevant behavior* of code, rather than a proof. 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 (as for PCC).

For policy specification other languages can be used as well. For instance, temporal logic formulae are widely applied for this purpose [8]. Also there is a number of XML-based languages for specification of access control policies, such as XACML [13]. However, while these languages suit well for describing security policies, they are less convenient for formal specification of the whole system, and in our framework it is essential to cover both these aspects. Therefore we chose a FSA-based language, which is suitable for specification of contracts as well as policies. However, it is worth noting that there is a mapping from temporal logic to FSA, which enables translating policies written as logic formulae into our FSA-based language.

7 Conclusion

In this paper we have proposed the notion of *security-by-contract*, a mobile contract that an application carries with itself. The key idea of the approach is that a digital signature should not just certify the origin of the code but rather bind together the code with a contract. From this point of view, our framework essentially makes more concrete some ideas behind MCC. In particular, we use a high level specification language with features that simplify contract/policy matching and allow expressing realistic security policies. Also our matching algorithm is improved for efficiency as it is intended for use on such resource-critical devices as mobiles. For this reason we first perform easier checks of sufficient criteria before performing a complete check (see Alg. 4). Other optimizations are also discussed. These features also differentiate our approach from other frameworks for modeling resource contractualisation, such as [18].

The contributions of the paper are threefold. 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 tentative structure for a contractual language. Finally, we have proposed a number of algorithms for one of the key steps in the life-cycle process: the issue of *contract-policy matching*.

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.

References

1. Building bug-free O-O software: An introduction to Design by Contract(TM). Availabe at `http://archive.eiffel.com/doc/manuals/technology/contract/`.
2. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
3. D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for Language Inclusion Using Simulation Relations. In *Proceedings of CAV'91: 3rd International Workshop on Computer Aided Verification*, pages 329–341. Springer, 1991.
4. K. Etessami. A hierarchy of polynomial-time computable simulations for automata. In *Proceedings of CONCUR'02*, pages 131–144. Springer-Verlag, 2002.
5. K. Etessami, T. Wilke, and R. Schuller. Fair Simulation Relations, Parity Games, and State Space Reduction for Buchi Automata. In *Automata, Languages and Programming, 28th international colloquium*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707. Springer, 2001.
6. L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, 1997.
7. L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.
8. K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Software Tools for Tech. Transfer*, 2004.
9. T. Henzinger, O. Kupferman, and S. Rajamani. Fair Simulation. In *Proceedings of CONCUR'97*, pages 273–287. Academic Press, Inc., 1997.
10. M. Jurdzinski. Small Progress Measures for Solving Parity Games. In *17th Symposium on Theoretical Computer Science (STACKS)*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2000.
11. Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Computer Aided Verification*, pages 97–109, 1993.
12. J. R. Michener and T. Acar. Managing System and Active-Content Integrity. *IEEE Computer*, 33(7):108–110, 2000.
13. T. Moses. eXtensible Access Control Markup Language (XACML) version 1.0. Technical report, OASIS, 2003.
14. G. C. Necula. Proof-Carrying Code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
15. G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. *SIGPLAN Not.*, 39(4):612–625, 2004.
16. R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and 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*, pages 23–30, New York, NY, USA, 2001. ACM Press.
17. R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-Carrying Code: a Practical Approach for Safe Execution of Untrusted Applications. *ACM SIGOPS Operating Systems Review*, 37(5):15–28, 2003.
18. N. Le Sommer. Towards Dynamic Resource Contractualisation for Software Components. In *International Working Conference on Component Deployment (CD 2004)*, volume 3803 of *Lecture Notes in Computer Science*, pages 129–143. Springer, 2004.
19. D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *IEEE Symposium on Security and Privacy*, 1998.
20. A. Zobel, C. Simoni, D. Piazza, X. Nuez, and D.Rodriguez. Business case and security requirements. Public Deliverable D5.1.1, EU Project S3MS, Report available at www.s3ms.org, October 2006.