S3MS Security of Software and Services for Mobile Systems

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
dragoni@dit.unitn.it

EUROPKI'07 - Mallorca, Balearic Islands, Spain

* Research partly supported by the project EU-IST-STREP-S3MS (http://www.s3ms.org)

Talk Outline

- Introduction
- Security-By-Contract Framework
  - Stakeholders
  - Contract and Policy
  - Application/Service Life-Cycle
- Contract-Policy Matching
  - Problem
  - Algorithm

Lack of Applications for Mobile Devices

- 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
- For instance, on high-end mobile phones we cannot even remotely find the amount of *third party software* that was available on our old PC

A Reason: Security Model for Mobile Code

- One of the reasons for this lack of applications is also the security model adopted for mobile phones.

- The current security model approach (for instance, the JAVA MIDP 2.0) 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 an appropriate trust domain.

Trust Relationship Problem (1)

The problem with trust relationship, i.e. digital signatures on mobile code, is twofold:

1. At first we can only reject or accept the signature.
   This means that interoperability 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 a platform and an application for paying parking meters is loaded, the user cannot block the application from performing large payments.

2. 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.

Trust Relationship Problem (2)

- From the point of view of the code consumers, they must essentially accept the code "as-is" without the possibility of making informed decisions.
  - 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 producers, 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.

Trust Relationship Problem (2)

- From the point of view of the code consumers, they must essentially accept the code "as-is" without the possibility of making informed decisions.
  - 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 producers, 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.

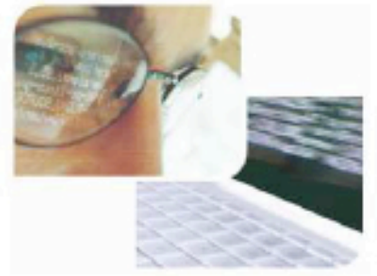
The consequence is that injecting an application in the mobile market is a time consuming operation as developers must essentially convince the operators that their code will not do anything harmful.

SxC Framework: Stakeholders

The Security-by-Contract framework is essentially shaped by three groups of stakeholders:

1. mobile operator
2. service provider and/or developer
3. mobile user

Operator



User

Mobile Platform Policy

Service Developer/Provider

Application Contract

Contract-Policy Matching

SxC Framework: Contract

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

SxC Framework: Contract

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

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).

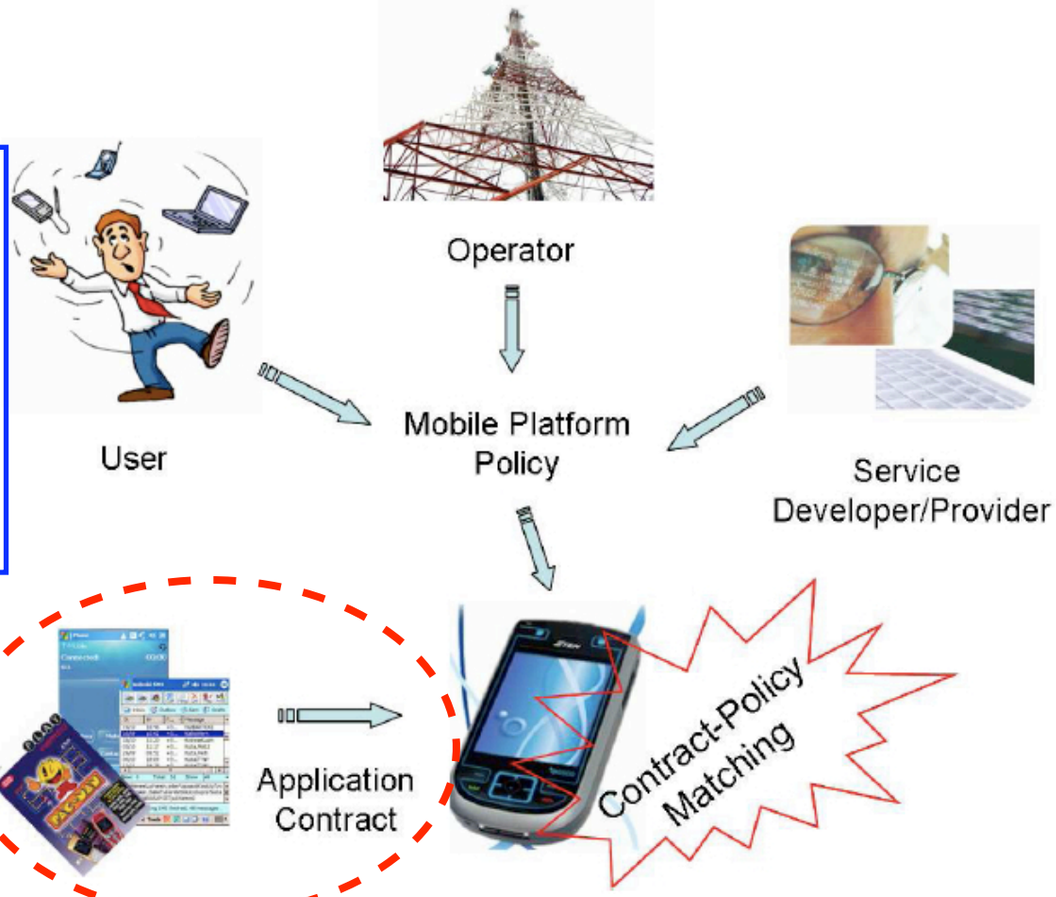
What's in a code's contract?

- security features of application
- (security) interactions with its host platform
- proof-of-compliance that code satisfies contract

SxC Framework: Contract

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

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).



What's in a code's contract?

- security features of application
- (security) interactions with its host platform
- proof-of-compliance that code satisfies contract

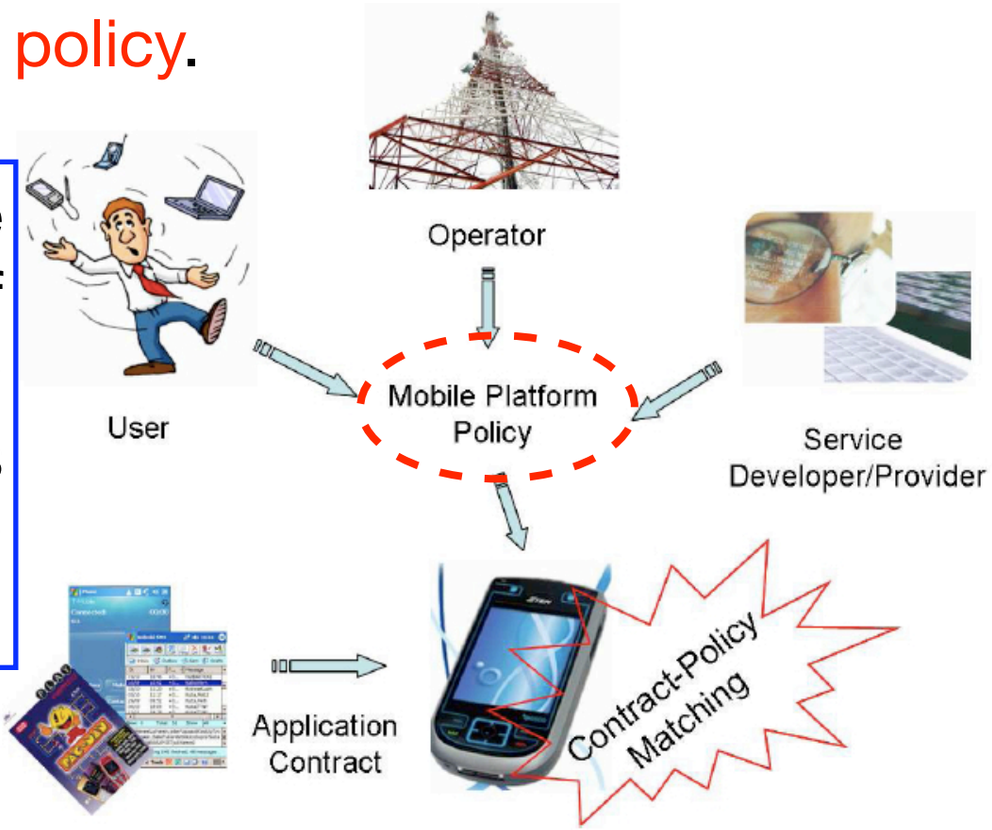
By signing the code the developer certifies that the code complies with the stated claims on its security-relevant behavior.

SxC Framework: Policy

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.

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).



What's in a platform's policy?

- ➡ platform contractual requirements on application
- ➡ 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

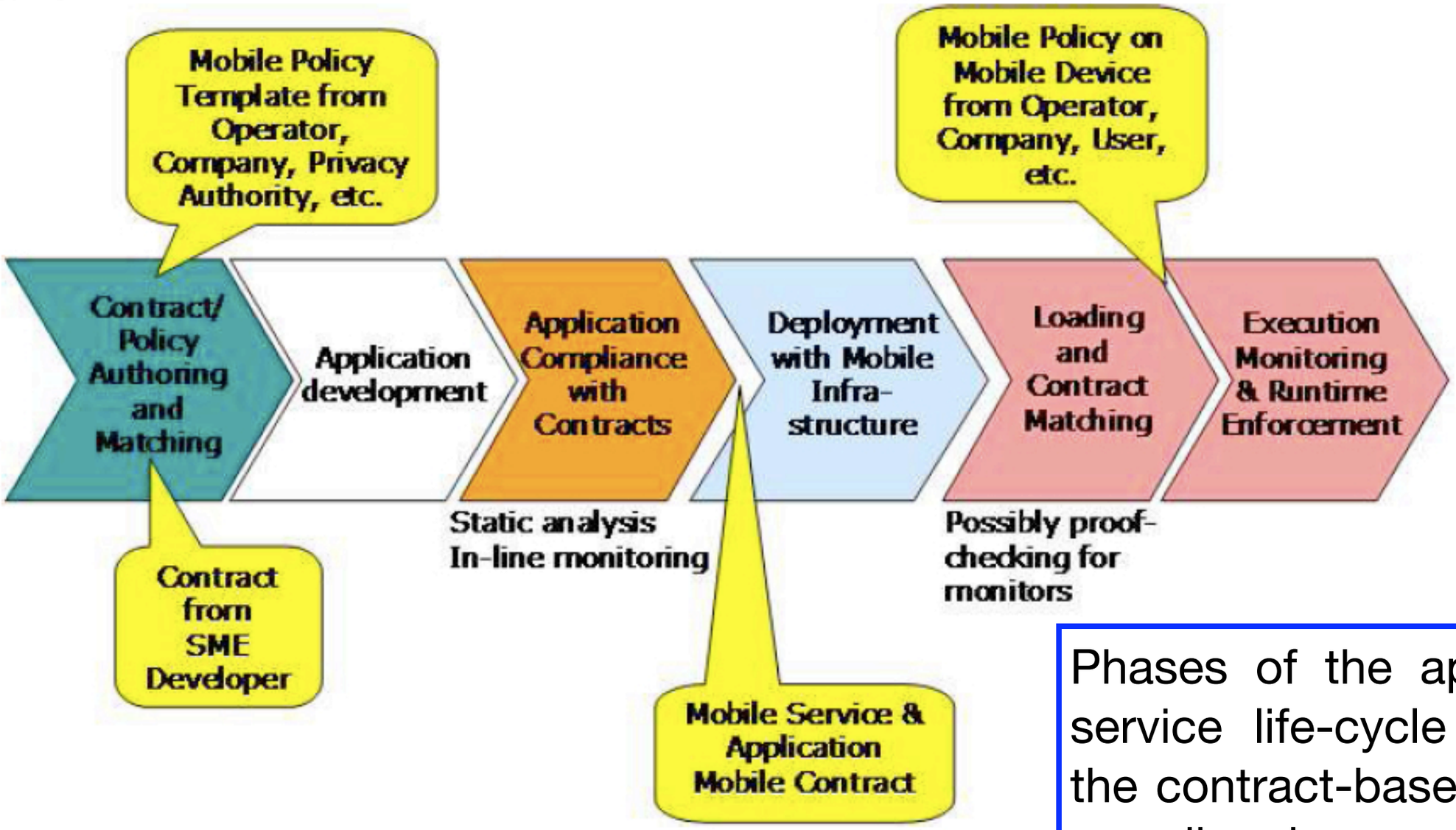
How a Contract/Policy Should Look Like?

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 must close all files that it opens
9. The application only receives SMS messages on a specific port

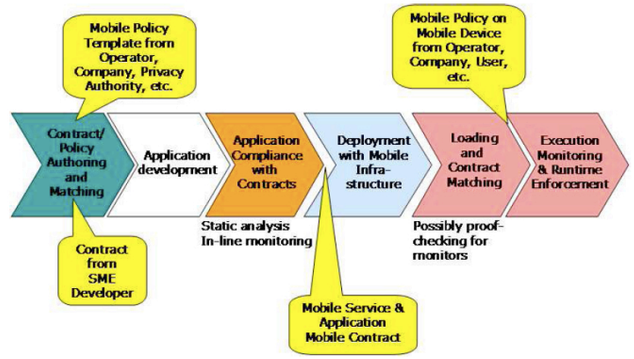
...

Application/Service Life Cycle (1)

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.



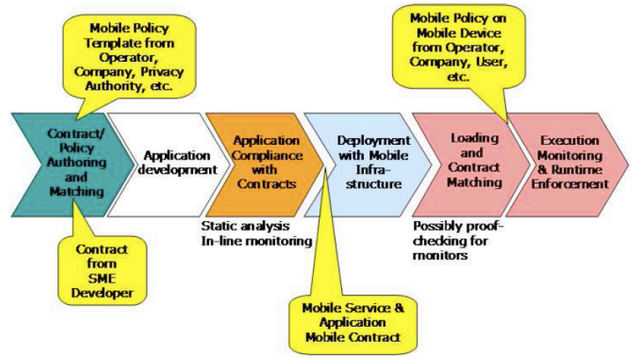
Phases of the application/ service life-cycle in which the contract-based security paradigm is present.



Application/Service Life Cycle (2)

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.

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

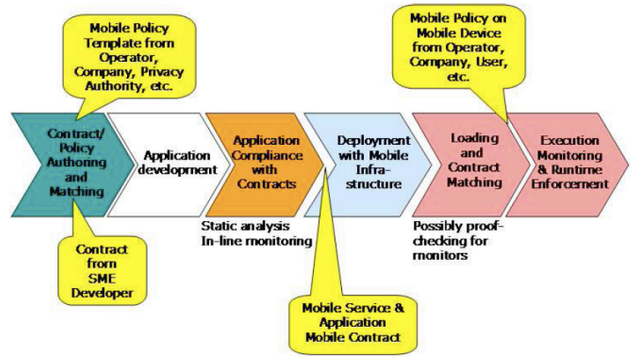


Application/Service Life Cycle (2)

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.

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

- Enforcing at level (I) can be achieved by appropriate design rules and require developer support

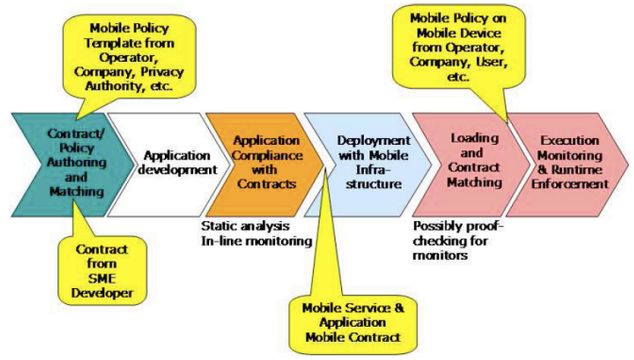


Application/Service Life Cycle (2)

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.

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

- 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)

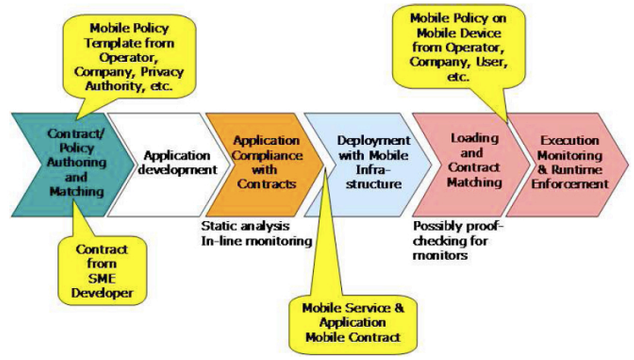


Application/Service Life Cycle (2)

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.

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

- 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



Contract/Policy Matching

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*
- *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 behaviour of the application that satisfies its contract also satisfies the platform's 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.

Contract Specification

A single contract/policy is specified as a list of *disjoint* rules 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>

Contract Specification

A single contract/policy is specified as a list of *disjoint* rules 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>

Scope definition reflects at which scope the specified contract will be applied.

OBJECT: the obligation must be fulfilled by each object of a given type.

SESSION: the obligation must be fulfilled by each run of the application separately.

MULTISESSION: the obligation must be fulfilled by all runs of the application as a whole.

Contract Specification

A single contract/policy is specified as a list of *disjoint* rules 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>

The tag RULEID identifies the area of the contract (which security-relevant actions the policy concerns, for example "files" or "connections").

Contract Specification

A single contract/policy is specified as a list of *disjoint* rules 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>

The tag RULEID identifies the area of the contract (which security-relevant actions the policy concerns, for example "files" or "connections").

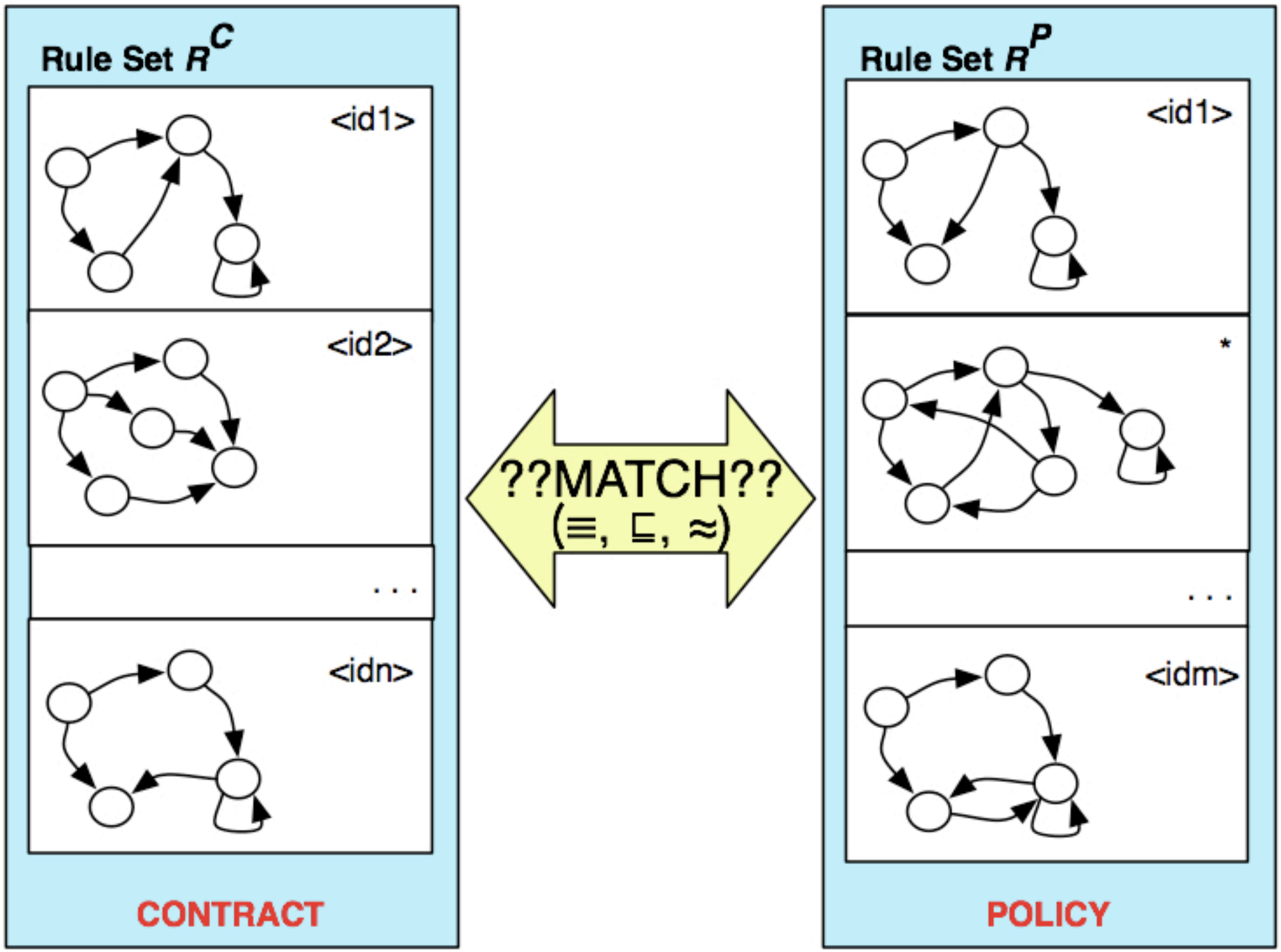
The <formal specification> part of a rule gives a rigorous and not ambiguous definition of the semantics of the rule.

Abstract Constructs

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

- [Combine Operator $\oplus$] Spec $= \oplus_{i=1,\ldots,n}$Spec$_i$
  It combines all the rule formal specifications Spec$_1$, . . ., Spec$_n$ in a new specification Spec.
- [Simulate Operator $\approx$] Spec$^C \approx$ Spec^P
  It returns 1 if rule formal specification Spec^C simulates rule formal specification Spec^P, 0 otherwise.
- [Contained-By Operator $\sqsubseteq$] Spec$^C \sqsubseteq$ Spec^P
  It returns 1 if the behavior specified by Spec^C is among the behaviors that are allowed by Spec^P, 0 otherwise.

Contract-Policy Matching Problem



Matching Algorithm (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)

Matching Algorithm (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: $\left\langle \mathcal{R}^C_{SESSION}, \mathcal{R}^C_{MULTISESSION}, \{\mathcal{R}^C_{class}\}_{class \in \zeta^C} \right\rangle \Leftarrow \text{Partition}(\mathcal{R}^C)$
2: $\left\langle \mathcal{R}^P_{SESSION}, \mathcal{R}^P_{MULTISESSION}, \{\mathcal{R}^P_{class}\}_{class \in \zeta^P} \right\rangle \Leftarrow \text{Partition}(\mathcal{R}^P)$
3: **if** MatchRules$(\mathcal{R}^C_{SESSION}, \mathcal{R}^P_{SESSION})$ **then**
4: **if** MatchRules$(\mathcal{R}^C_{MULTISESSION}, \mathcal{R}^P_{MULTISESSION})$ **then**
5: **for all** $class \in \zeta^P$ **do** // *for all classes in policy*
6: **if** MatchRules$(\mathcal{R}^C_{class}, \mathcal{R}^P_{class})$ **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: $\mathcal{R}_{SESSION} \Leftarrow \{r \in \mathcal{R} \mid \text{Scope}(r) = \text{SESSION}\}$
2: $\mathcal{R}_{MULTISESSION} \Leftarrow \{r \in \mathcal{R} \mid \text{Scope}(r) = \text{MULTISESSION}\}$
3: **for all** $class \in \zeta$ **do** // *for all classes in contract/policy*
4: $\mathcal{R}_{class} \Leftarrow \{r \in \mathcal{R} \mid \text{Scope}(r) = \text{OBJECT} < class >\}$
5: **end for**

Matching Algorithm (2)

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

Matching Algorithm (3)

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** $\exists\ (\mathrm{ID}^C,\ \mathrm{Spec}^C) \in L^C \wedge\ \mathrm{ID}^C = \mathrm{ID}^P$ **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$\left(\left(*,\ \oplus_{(\mathrm{ID}^C,\ \mathrm{Spec}^C) \in L^C} \right), (*,\ \mathrm{Spec}^P) \right)$
13: **end if**

Matching as Language Inclusion

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**

Matching as Language Inclusion

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**

on-the-fly

Conclusion

The contributions of the paper are threefold:

1. First, we have proposed the security-by-contract framework providing a description of the overall life-cycle of mobile code in this setting.

2. Then we have described a tentative structure for a contractual language.

3. 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.

Conclusion

The contributions of the paper are threefold:

1. First, we have proposed the security-by-contract framework providing a description of the overall life-cycle of mobile code in this setting.

2. Then we have described a tentative structure for a contractual language.

3. 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.

THANKS!

Appendix A - Definitions

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:

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

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.

3. 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. 2 we might reject "good" applications that are however too difficult or too complex to perform. On the other hand, Def. 3 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).

Appendix B - Abstract Constructs

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 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 $Spec$.