Security-by-Contract (S×C) for Software and Services of Mobile Systems[*]

N. Dragoni[1] F. Massacci[1†] P. Mori[2] C. Schaefer[3] T. Walter[3] E. Vetillard[4]

1. University of Trento
2. Institute of Informatics and Telematics, CNR, Italy
3. DoCoMo Euro-Labs, Germany
4. Trusted Logic, France

Submitted for consideration as a Chapter for the "At your service" book by MIT"

Abstract

In this chapter we propose the *security-by-contract* (SxC) framework and its technological implementation for trusted deployment and execution of communicating mobile applications in heterogeneous environments. The objective is to build the basis for the opening of the software market of nomadic devices (from smart phones to PDA) to third party applications.

The intuition of SxC is that applications should come equipped with a security contract (as in programming-by-contract [4]). In a nutshell, a contract describes the security relevant interactions that the mobile application could have with the mobile device. The contract should be accepted by the platform (if compatible with the policy) at deployment time, and its enforcement guaranteed either by static analysis at development time or by monitoring at run time.

This paradigm will not replace, but enhance today's security mechanisms, and will provide a flexible, simple and scalable security mechanism for future mobile systems.

1 Introduction

The paradigm of pervasive services [3] envisions a nomadic user traversing a variety of environments and seamlessly and constantly receiving services from other portables, handhelds, embedded or wearable computers. Bootstrapping and managing security of services in this scenario is a major challenge.

We argue that the challenge is bigger than the "simple" pervasive service vision because it does not consider the possibilities that open up when we realize that the smart phone in

[*]This chapter describes some of the research results developed in the course of the EU IST FP6 STREP Project S^3MS (*Security of Software and Services for Mobile Systems*), www.s3sm-project.org.

[†]Contact author: Dip. Informatica e Telecomunicazioni - via Sommarive 14, Povo (Trento), Italy. Fabio.Massacci@unitn.it.

our pocket has *already* more computing power than the PC encumbering our desk 15 years ago.

Current pervasive services, including context-aware services, do not exploit the computational power of the mobile device. Information is provided to the mobile user anywhere but the computing infrastructure is centralized [11]. Even when it is decentralized to increase scalability and performance [7, 5], it does not exploit the devices' computing power.

We believe that the future of pervasive services will be shaped by *pervasive client downloads*. When traversing environments the nomadic user does not only invoke services according a web-service-like fashion (either in push or pull mode) but also download *new* applications that are able to exploit its computational power in order to make a better use of the unexpected services available in the environment.

A tourist landing in a historical city might download at the airport a *tourist guide* application that can route her rented car to those touristic hotspots that are among her particular interests. The application is being configured with mentioned touristic hotspots and, in order to determine the route to those hotspots, the application needs to interact with the car's navigation system to determine the current location and to update the route planning (but only if confirmed by the driver), and might send travel tips to selected driver's companions.

This is a pervasive service download because it offers local services, we want it exactly where we need it (e.g. via a bluetooth link at the airport), without bothering a long and frustrating web search before departing, we want to use it on our mobile or PDA without connecting to a remote route planner each and every time.

Such scenarios create new threats and security risks on top of the "simple" pervasive service invocation because it violates the heart of the security model of mobile software in Java [10] and .NET [21, 14] [14]:

- A pervasive download is essentially untrusted code whose security properties we cannot check and whose code signature (if any) has no degree of trust[1];
- According to the classical security model it should be sandboxed, its interaction with the environment and the devices own data should be limited;
- Yet this is against the whole business logic, as we made this pervasive download precisely to have lots of interaction with the pervasive environment!
- In almost all cases this code will be trustworthy, being developed to exploit the business opportunities of pervasive services.

Another example is services for mobile workers in an ubiquitous environment who have to share not only data among each other but, if from different enterprizes, may share their applications as well [17]. Web browser plug-ins, web clients, and collaborative tools are other examples.

The current security model is highly unsatisfactory: if we download a client in order to use a service we either trust it fully or not at all. In contrast we need a flexible mechanism that allows the owner of the mobile platform to control which actions of a given application

[1]Most software is from small companies which cannot afford the expences necessary to obtain an operator's certification and thus will not run as trusted code.

Criteria	**Static Analysis**	**In-lining**	**Runtime**
Works with existing devices	$\surd$	?	$\times$
Works with existing applications	?	$\times$	$\surd$
Does not modify applications	$\surd$	$\times$	$\surd$
Offline proof of correctness	$\surd$	$\surd$	$\times$
Load-time proof of correctness	$\times$	$\surd$	$\times$
May depend on run-time data	$\times$	$\surd$	$\surd$
Does not affect runtime performance	$\surd$	?	$\times$

Table 1: Enforcement Technology Strengths and Weaknesses

are allowed on a platform. The enforcement of the platform security policies can be taken at different stages of the mobile application's life-cycle. Each stage will have different functionality constraints:

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

(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* [23, 20] by developers and operators followed by a *trusted signature*) or as a combination of pre and post-loading operations (e.g., through *in-line monitors* [9] and *proof carrying code* [2, 18]); (IV) can be implemented by *run-time checking* [15]. All methods have different technical and business properties. For example, from an operator's view point:

- working on existing devices would rule out run-time enforcement, and favour static analysis. 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.
- An operator who wants to be able to run existing applications would prefer run-time enforcement.

The Table 1 shows some of possible strengths and limitations of each different technology.

The basic idea behind the S^3MS project (www.s3ms.org) is to put together these enforcement technologies in order to build the Security-by-Contract framework.

1.1 Our Vision

We advocate the notion of *Security-by-Contract (S×C)* as a mechanism to make it possible the trustless dissemination of trustworthy code in the pervasive service scenario that we have just described. The key idea behind S×C is that a digital signature should not just certify the origin of the code but rather bind together the code with a contract.

Loosely speaking, a *contract* is just set of rules describing the security behavior of the *mobile application* with its host platform. Some examples of security relevant rules are silently initiate a phone call or send an SMS, memory usage, secure and insecure web connections, access to user's address book, confidentiality of application data, constraints on access from other applications already on the platform. We discuss informally the syntax in Section 2 and refer to [1] for details. A mobile application can take a number of forms such as a Java plug-in for a browser or a .NET assembly to be installed on the mobile phone virtual machine. Essentially it is a piece of code that we download on our mobile platform in order to use some services in the surrounding environment.

To understand how this apparently minor addition can dramatically change the reality of pervasive services let's consider the life-cycle of a mobile client. Figure 1 summarizes the phases of the application/service life-cycle in which the contract-based security paradigm is present. Still, mobile systems will be more customizable and trustworthy, and 3rd-parties services can be used securely.

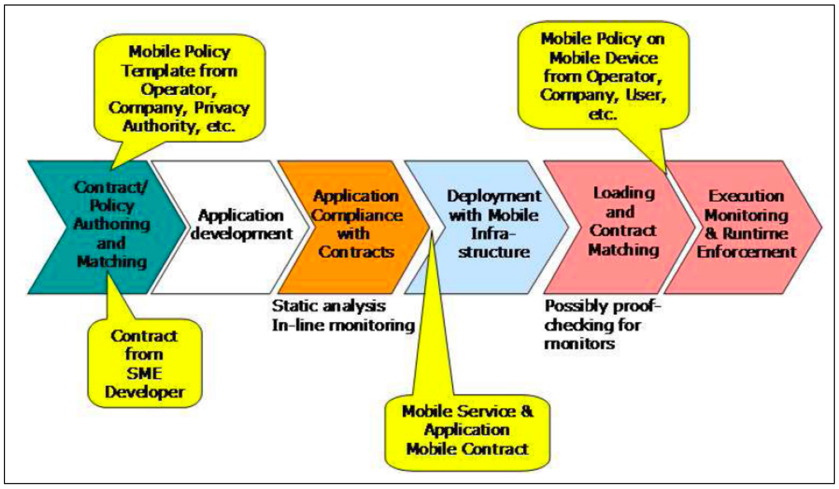

Mobile Policy Template from Operator, Company, Privacy Authority, etc.
Mobile Policy on Mobile Device from Operator, Company, User, etc.
Contract/ Policy Authoring and Matching
Application development
Application Compliance with Contracts
Deployment with Mobile Infra-structure
Loading and Contract Matching
Execution Monitoring & Runtime Enforcement
Static analysis In-line monitoring
Possibly proof-checking for monitors
Contract from SME Developer
Mobile Service & Application Mobile Contract

Figure 1: Application/Service Life-Cycle

At development time a service provider could check its downloadable client (the mobile application) against a contract template that is provided by operators or other service brokers. *Static analysis* could be efficiently used to check that a mobile application satisfies the template contract. We discuss this possibility in Section 3.

At the other end of the life-cycle, end users, operators or companies managing the platform may deploy a *platform policy*, describing the security behavior expected (or at least desired) from all applications to be downloaded on the platform. At this stage we don't need a separate language and contracts can be re-used also for policies.

At deployment time (i.e. when clicking the "download" button) the contract should be accepted by the platform depending on the actual security policy that the platform has. This *matching* step must be supported to avoid that users wishing to download a service client simply click through their way by accepting all possible bad actions. We discuss the key issues behind matching in Section 4.

During the execution of the mobile application the platform might opt for *run-time* *monitoring* the application anyhow (for example in cases where contract matching failed or there is no trusted signature binding together code and contract. We discuss how this can be done in Section 5.

The whole architecture of S×C can itself be represented as a service oriented infrastructure where each of the above mentioned services (matching, static analysis, etc.) can be represented as a service. We analyze the security functionalities and the threat models of S×C as a SOA in Section 6.

Finally we conclude our chapter with a some remarks on the role of security in the broader outlook of S×C.

2 What is a Security Contract

As we have said, a security contract is just a claim on the security relevant actions of the mobile applet (i.e. the client downloaded on the platform in the pervasive services scenario).

In the S×C framework, a single contract/policy can be seen as a *list of disjoint rules* (for instance rules for connections, rules for Personal Identification Module, PIM, and so on), where each 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”).

Intuitively the session scope define behavior that can belong to a single execution of the mobile code. The multi-session scope define rules with “memory” that describe properties that ought to be true across subsequent executions of the code.

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 **??**.

The `<formal specification>` part of a rule gives a rigorous and not ambiguous definition of the behaviour (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). In the framework of the S^3MS project the industry partners of the consortium have done a careful requirements analysis in order to capture the key business needs (see www.s3ms.org). Some of the examples of desired security “contracts” for mobile applications are detailed below:

- send no more than a given number of messages in each session;

- only loads each image from the network once;
- do not initiate calls to international numbers;
- only makes call to fixed premium SMS numbers;
- do not send MMS messages.
- connects only to its origin domain.
- The length of a SMS message sent does not exceed the payload of a single SMS message.

and so on.

Such requirements have been further distilled in the minimal characteristics of contractual features that should be captured and demonstrated:

- *permitting or prohibiting the activation or deactivation of a security relevant service* (e.g. opening a communication, sending an SMS, starting an application, modifying the address book etc.)
- *presence of past events as a pre-requisite for allowing another present event* (e.g. the user confirmation before an SMS is sent or a image is downloaded)
- *cumulative accounting of events* (e.g. downloading at most 5MB of images and sending at most 3 SMS during a day time).

The next step is of course the *specification of the security-by-contract language* that is able to capture the above features. We refer to [1] for details and here we show informally some examples in order to give a taster.

Example 2.1 (Requirements G-37 ([16]) *ConSpec: "The MIDlet only establishes HTTP connections".*

```
SCOPE Session
SECURITY STATE
BEFORE javax/microedition/io/Connector.open(java/lang/String url)
PERFORM
  url.startsWith("http:") -> { skip; }
BEFORE javax/microedition/io/Connector.open(java/lang/String url, int mode)
PERFORM
  url.startsWith("http:") -> { skip; }
BEFORE javax/microedition/io/Connector.open(java/lang/String url, int mode,
       boolean timeouts)
PERFORM
  url.startsWith("http:") -> { skip; }
```

Example 2.2 *Let us consider an application's contract which includes two rules: one for using HTTPS network connections and the other for restricting sending messages.*

- The application only uses HTTPS network connections.

- No messages are sent by the application.

The corresponding ConSpec specification follows:

```
MAXINT 10000 MAXLEN 10

RULEID HIGH_LEVEL_CONNECTIONS
SCOPE Session
SECURITY STATE
boolean opened = false;
BEFORE javax.microedition.io.Connector.open(string url)
PERFORM
  url.startsWith("https://") && !opened  -> { opened = true; }
  url.startsWith("https://") && opened -> { 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; }
```

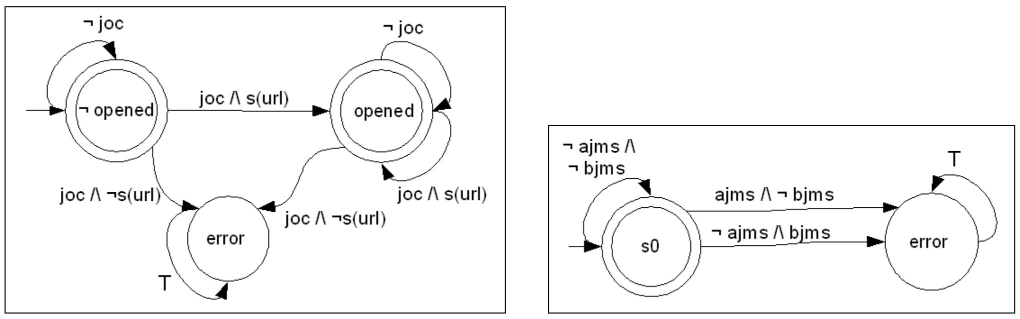
3 Static Analysis

Static analysis provides a way to verify that an application's code complies to the application's declared contract, but it needs to be performed before actually deploying the application, as the technology is too complex to be run on the target device. Static analysis therefore needs to be used in conjunction with other technologies, which will help relaying the proof provided by a static analysis tool to the actual device, such as:

Digital signatures. The most classical way to use static analysis is to make the use of a static analysis tool one of the conditions for the generation of digital signature for an application, for instance as part of a systematic certification program.

Proof-carying code. The static analysis tool is used to build a correctness proof, which is sent over with code and verified on the device. This technology relies on the fact that it is much easier to verify a proof than to actually infer the information required to build it. It is for instance used by the Java bytecode verifier used in all Java ME devices.

On-the-fly analysis. The static analysis tool runs during the deployment process. This of course supposes that the tool can run fast enough, and that the deployment process



(a) Automaton 1 (b) Automaton 2

Abbreviations for JAVA APIs:

joc	$\doteq$	io.Connector.open(url)
$s(url)$	$\doteq$	url.startsWith("https://")
$ajms$	$\doteq$	after MessageConnection.send(message)
$bjms$	$\doteq$	before MessageConnection.send(message)
s_0	$\doteq$	initial state, the system is staying in this state until it sends the message

Figure 2: Automata for the Contract of Ex. 2.2

occurs over a secure connection, in order to guarantee that the application actually originates from a server that uses the appropriate static analysis tools.

Such technologies are not the main focus of the S^3MS project, which looks at the core technologies required for the use and enforcement of contracts, while minimizing the impact on the infrastructure itself. The fact that several technologies can be used for enforcement is actually an advantage for S×C, as it reduces the constraints on the deployment of the technology. In particular, the fact that S×C can rely on digital signatures is a very strong point, because all mobile application frameworks already support digital signatures as a way to guarantee that an application has been properly certified, so the adaptation of the device for supporting S×C' static analysis can be minimal. In some cases, it simply consists in the use of a specific public key.

There is a strong incentive for application developers to use static analysis tools during the development process: static analysis tools are by essence imperfect, because they need to reason on an approximation of the application's behavior. Therefore, static analysis tools are often unable to prove that an application verifies a property, although it actually verifies it. Static analysis tools usually make assumptions about the developer's behavior, and each tool favors some programming habits, that happens to suit the way in which the tool is built. Here are a few examples of such habits:

Programming patterns. Most static analysis tools are able to get better results if the developers provide code that matches specific programming patterns. Such patterns are usually well known, and often correspond to the typical uses of APIs.

Program annotations. Some static analysis programs are able to use additional information entered by the developer as annotations. This information is used to make the analysis more precise, usually at points where the algorithms have difficulties inferring the information by themselves. Annotations can also be a way to make the analysis of a program faster.

Both programming patterns and program annotations can be useful with the algorithms used in S×C.

To show the feasibility of the approach in the S^3MS project we have extended to S×C an industrial tool, that has been used to enforce security policies defined by operators. It is based on a rather simple analysis, based on abstract interpretation. The analysis is interprocedural, but it relies heavily on a method-local analysis. In particular, the tool infers very limited information about sequences of operations across methods. The Java ME static analysis tool is therefore not intended to be the only code-contract compliance verification method. Its objective is here to verify as many contract properties, in order to reduce the runtime overhead. Initially, the Java static analysis tool has been designed for the verification of a fixed set of rules, most of them being hard-coded. Then, the purpose of this section is to expose the development of generic verifiers in the tool. *Generic verifiers*, which are well suited for contract verification, satisfy the high-level business requirements specified in S×C. They can be used in several scenarios, either off-line, before the deployment, or on-line, during the deployment (if the performance level is sufficient).

In the context of the S^3MS project, the static analysis implemented by Trusted Logic is able to determine a Java virtual machine state (stack, frame, heap) as well as any action on this state (API method invocation or heap accesses). On the opposite, it is not able to enforce temporal security requirements(*e.g.*, delay between operations). Thus, *API Usage*, *API Usage Restrictions* and *Mandatory Sequences* can only be partly verified by the static analysis.

API Usage (Restrictions) This is the simple category of generic verifiers to implement in the Java static analysis tool. The characteristic of such a verifier should be as follows:

- It requires to register (for instance through Java listener mechanism) to a fixed list of API method events identified by the method signature.
- It should be able to compare expected (contract) and observed (static approximation) registered method arguments, possibly ignoring some of the arguments: in the `Connector.open("http://"*, READ, *)` expected expression, the `timeout` parameter could take any value.
  In addition, note that the argument comparison is delegated to the abstract domain of each type of value and also the precision of the contract may differ from a value to another. In the context of the S×C project, the tool should at least support these expressions:

  Boolean Domain. A requirement on boolean values could be expressed by a concrete value in tuple $\{true, false\}$ or by any value $(*)$.

  Integer Domain. A requirement on integer values could be expressed by a concrete value in each associated range (*e.g.*, $[-128, 127]$ for byte) or by any value $(*)$.

Floating Point Domain. Floating point values are not supported by ConSpecsimilar to integer values (a concrete value or any value).

String Domain. String requirements are the most important elements in the context of Java mobile application certification since strings are used for any sensitive operations in the application such as connection URLs. Thus, any standard string comparisons in Java should be supported both by ConSpec and the static analysis tool. A minimal list of string requirements must contains: string prefix(es) and string suffix(es) requirements, string's equality, string's length.

Reference Domain. About references, requirements should be limited to the tuple $\{null, < not\_null >, *\}$. In addition, the $< not\_null >$ requirement may specify a specific instance type for simple references, while for arrays it may specify an array element type, a dimension and an array length.

Mandatory Sequences Some of the sequencing requirements could be expressed in a generic manner too. In the context of the S×C project, the Java static analysis tool only supports sequencing requirements that are limited to sequences of events local to a method body. Thus, it could explore the application control flow graph regardless of the concurrency issues on mobile applications.

A generic verifier handling sequencing requirements has the following characteristics:

- It should be able to extract from the ConSpec policy the association state variable updates and related events.
- Then, it should construct in a generic manner a control flow graph branch that should be matched to the analysed one.

Example 3.1 *Here is provided an example of the definition of the Mobius security requirement G-27 ([16]) in ConSpec: "The application does not send messages in a loop".*

```
SCOPE Session
SECURITY STATE
  bool loop = false;
BEFORE enterloop // Unexisting event modifier
PERFORM
  TRUE -> { loop = true; }
BEFORE exitloop //Unexisting event modifier
PERFORM
  TRUE -> { loop = false; }
BEFORE javax/wireless/MessageConnection.send(javax/wireless/Message msg)
PERFORM
  !loop -> { skip; }
```

4 Contract-Policy Matching

In order to define the matching between a contract we abstract from a particular formal specification[2], 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.

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

- [Combine Operator $\oplus$] $\mathrm{Spec} = \oplus_{i=1,\ldots,n}\mathrm{Spec}_i$
  It combines all the rule formal specifications Spec_1, …, Spec_n in a new specification Spec.

- [Simulate Operator $\approx$] $\mathrm{Spec}^C \approx \mathrm{Spec}^P$
  It returns 1 if rule formal specification Spec^C simulates rule formal specification Spec^P, 0 otherwise.

- [Contained-By Operator $\sqsubseteq$] $\mathrm{Spec}^C \sqsubseteq \mathrm{Spec}^P$
  It returns 1 if the behaviour specified by Spec^C is among the behaviours 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 4.1 $\mathsf{Traces}\,(Spec_1 \oplus Spec_2) = \mathsf{Traces}\,(Spec_1) \cup \mathsf{Traces}\,(Spec_2)$

Property 4.2 $Spec_1 \sqsubseteq Spec_2 \Leftrightarrow \mathsf{Traces}\,(Spec_1) \subseteq \mathsf{Traces}\,(Spec_2)$

Property 4.3 $Spec_1 \approx Spec_2 \Rightarrow \mathsf{Traces}\,(Spec_1) \subseteq \mathsf{Traces}\,(Spec_2)$

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:*

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

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 satisfies the policy.*

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

[2]Interested readers can find in [8] an example of exploitation of our matching algorithm for automata-based rule specifications

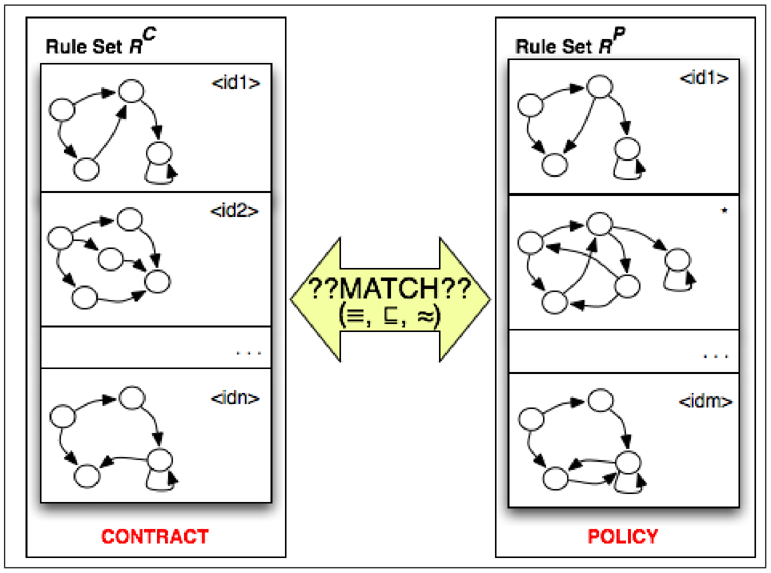


Figure 3: Contract-Policy Matching Problem

By applying Def. 4.2 we might reject "good" applications that are however too difficult or too complex to perform. On the other hand, Def. 4.3 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).

The algorithm is *generic* since it does not depend on the formal model adopted for specifying the semantics of rules. Namely, it is defined by means of the abstract constructs discussed in the previous Section. Therefore, to actually exploit the algorithm it will be sufficient to have an implementation of such constructs in the formal language adopted for specifying rules.

As shown in Fig. 3, the 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 whether or not $\mathcal{R}^C$ "matches" $\mathcal{R}^P$.

Algorithm 1 lists the pseudo 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. As already mentioned, this partition is necessary because in the S×C 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 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. The match is checked 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 2). Since the rules of the two input sets $\mathcal{R}^C$ and $\mathcal{R}^P$ have

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: Partition $\mathcal{R}^C$ according to the SCOPE of the rules
 2: Partition $\mathcal{R}^P$ according to the SCOPE of the rules
 3: if rules with SCOPE SESSION do match (call to the MatchRules function) then
 4:    if rules with SCOPE MULTISESSION do match (call to the MatchRules function) then
 5:       for all classes in policy do
 6:          if rules with SCOPE OBJECT do match (call to the MatchRules function) then
 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)
```

the same scope, before starting the match the algorithm cleans $\mathcal{R}^C$ and $\mathcal{R}^P$ removing the tag 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 perform 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 and 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 3) checks the match between a set of pairs $\mathcal{L}^C = \left\langle \left(\mathrm{ID}^C_1,\ \mathrm{Spec}^C_1\right), \ldots, \left(\mathrm{ID}^C_n,\ \mathrm{Spec}^C_n\right) \right\rangle$ 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.

Matching is performed 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 2 MatchRules Function

```
Input: rule set $\mathcal{R}^C$, rule set $\mathcal{R}^P$ (both containing rules with the same SCOPE)
Output: 1 if $\mathcal{R}^C$ matches $\mathcal{R}^P$, 0 otherwise
 1: Remove tag SCOPE from all the elements of $\mathcal{R}^C$ and save the new list $L^C$
 2: Remove tag SCOPE from all the elements of $\mathcal{R}^P$ and save the new list $L^P$
 3: for all $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$ in $L^P$ do
 4:     if there exists a rule in $L^C$ that matches $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$ (call to the MatchSpec function) then
 5:         skip
 6:     else {may return $\emptyset$ for efficiency}
 7:         add the element $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$ to the list $L^P_{failed}$
 8:     end if
 9: end for
10: if $L^P_{failed}$ is empty then
11:     return(1)
12: else
13:     call MatchSpec with the combination of the contracts in $L^C$ and the combination of the policies in
        $L^P_{failed}$ and return the result
14: end if
```

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).

Algorithm 3 MatchSpec Function

```
Input: $L^C = \langle (\mathrm{ID}^C_1,\ \mathrm{Spec}^C_1), \ldots, (\mathrm{ID}^C_n,\ \mathrm{Spec}^C_n) \rangle$, $(\mathrm{ID}^P,\ \mathrm{Spec}^P)$
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($\mathrm{Spec}^C$) = HASH($\mathrm{Spec}^P$) 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:     $\mathsf{MatchSpec}\Big(\big(*,\ \oplus_{(\mathrm{ID}^C,\ \mathrm{Spec}^C) \in L^C}\big),\ (*,\ \mathrm{Spec}^P)\Big)$
13: end if
```

5 Run-time Enforcement on Java ME

Java 2 Micro Edition (J2ME) consists of three distinct layers, the Mobile Information Device Profile (MIDP), the Connection Limited Device Configuration (CLDC) and the

Kilo Virtual Machine (KVM), as shown in Fig. 4. Each of these layers provides a specific security support.

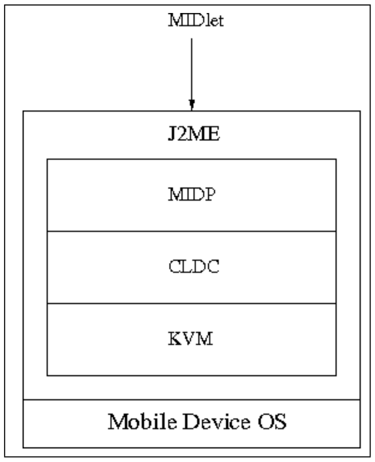


Figure 4: Java 2 Micro Edition run-time environment

The security support provided by the CLDC [24] concerns the low level and the application level security. To execute the MIDlet, the CLDC adopts a sandbox model, that requires that: the MIDlet has been preverified; the MIDlet cannot bypass or alter standard class loading mechanisms of the KVM; only a predefined set of APIs is available to the MIDlet; the MIDlet can only load classes from the archive it comes from (i.e. from the Jar file including it); and, finally, that the classes of the system packages cannot be overridden or modified.

The security support provided by the MIDP [12, 13] defines a set of protection domains, and pairs a set of permissions with each of these domains. Each MIDlet that runs on the device is bounded to one of these protection domains, and this determines the value of its permissions. A protection domain is assigned to a MIDlet depending on the who signed the MIDlet itself, and can be: Manufacturer, Operator, Identified Third Party and Unidentified Third Party. If the MIDlet is not signed, then is paired with the Unidentified Third Party protection domain. The permissions refer to the operations that the MIDlet can perform during its execution and the value that can be paired with them can be either *allowed* or *user*. As an example, the *javax.microedition.io.Connector.http* permission refers to HTTP connections. If the value is *allowed*, the permission is granted, otherwise a user interaction is required to enter the value of this permission.

5.1 S×C Security

The S×C framework enhances the J2ME standard security support by enforcing security policies at run-time, i.e. the last stage of the application life-cycle described in Fig. 1. The security policy is defined through the ConSpec language, as previously seen for contracts, and defines which security relevant actions the MIDlet can perform during its execution. In particular, the actions we are interested in are the methods of the J2ME core classes that perform interactions with the underlying device. As an example, opening a connection with a remote partner is considered a security relevant action, because the connection could be exploited by the MIDlet to send personal data to an unknown entity, while the conversion of an integer value into a string is a negligible action from the security point of view. The policy pairs each method with a set of conditions that must be satisfied before and/or after the execution of the method itself. For example, these conditions may concern the value of the method parameters or the value of some policy variables. With respect to the static analysis, the runtime enforcement can evaluate conditions that depend on input data. As an example, if we want that the security policy allows connections with remote servers only if the target URL begins with a given prefix, e.g. "http://www.google.it/", this control can be implemented by pairing a condition that checks the URL parameter value before the execution of the open connection method. This value could be obtained by the MIDlet as an input parameter, or as a result of a previous operation. The security policy can also take into account the state of the execution, or define dependencies among the execution of actions, i.e. it can define the order in which actions are performed. As an example, the security policy can state that only three HTTP connections can be opened at the same time, or that further HTTP connections cannot be opened after that an HTTPS connection has been opened.

With respect to the standard J2ME security support, the security policies supported by the S×C framework define finer granularity controls and an history-based monitoring of the MIDlet. As a matter of fact, the policy defines the sequences of operations that the MIDlet can execute. In this way, the right of the MIDlet to execute an action does not depend on the actions itself only, but also on all the other actions that have previously been executed by the MIDlet.

From the architectural point of view, the enforcement of a security policy during the execution of a MIDlet is performed through the integration in the J2ME architecture of two components: a Policy Decision Point (PDP), that evaluates the current security relevant action against the security policy, and a MIDlet monitoring component, that intercepts the security relevant actions performed by the MIDlet during its execution, invokes the PDP for the evaluation of the policy, and that enforces the decision taken by the PDP. This implies that the S×C runtime support works with existing MIDlets, but requires the upgrade of the software of mobile devices.

Several solutions can be possible to integrate the MIDlet monitor component in the J2ME architecture. As an example, the system calls that the KVM performs on the operating system of the underlying mobile device could be intercepted by the monitor and considered as security relevant actions. However, this solution has not been adopted because we are interested in monitoring the MIDlet behaviour at methods level. Moreover, since the set of system calls could be different on distinct mobile devices, defining the security policy in terms of system calls prevents the portability of the security policy.

Another solution is the one that exploits the permissions defined by MIDP. In this case, the monitor could be embedded in the MIDP component that evaluate the permission, that is invoked by the J2ME every time that an action that involves a permission is executed. As an example, let us suppose that the MIDlet requests to open an HTTP connection with a remote URL by exploiting the *javax.microedition.io.Connector* class of the MIDP. In this case, the value of the *javax.microedition.io.Connector.http* permission decides whether the HTTP connection can be established. However, this solution defines as security relevant actions only the ones that are also paired with a permission. Moreover, this solution does not allow to perform the test of the security policy after the execution of the security relevant action, because the MIDP permissions are evaluated only before the execution of the action.

The adopted solution is based on the modification of the J2ME platform. As a matter of fact, we choosed a subset of the methods of the API provided by MIDP and CLDC as set of security relevant actions, and the MIDlet monitoring component has been embedded in the J2ME architecture by modifying the source code of these methods. The modification simply consists in inserting the invocation of the PDP at the beginning and at the end of the source code that implements these methods. In this way, every method of the J2ME could, in principle, be defined as security relevant action. If the result of the invocation of the PDP is negative, i.e. the execution of the action is denied according to the security policy, a SecurityException error is thrown by the method. The MIDlet could be instrumented to manage this exception, and in this case it continues running, otherwise it fails. The resulting architecture is described in Fig. 5.

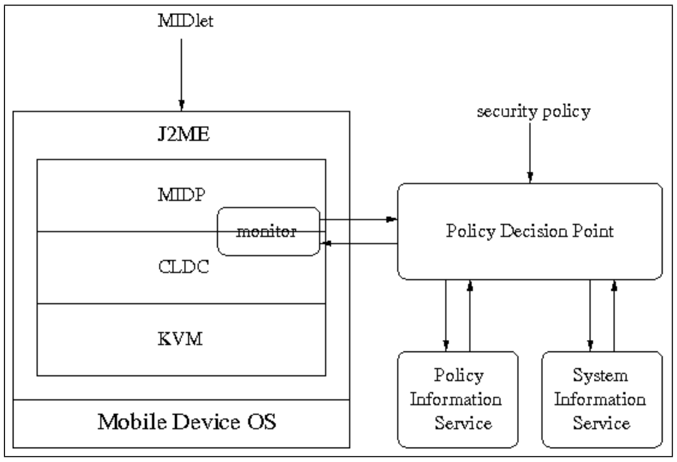


Figure 5: Java 2 Micro Edition run-time monitoring

The Policy Decision Point (PDP) is the component of the architecture that decides whether a given security relevant method can be performed in a given state according to the policy. The PDP is invoked by the MIDlet monitor twice for each security relevant method that the MIDlet tries to execute, and this invocation reports to the PDP the method full name, its parameters, the name of the MIDlet, an ID of the MIDlet and a flag that specifies whether the invocation has been made before or after executing the method.

To evaluates the security policy, the PDP could need the value of some policy variables. As an example, a policy could allow to open a further network connection only if this MIDlet has opened less than X connections. In this case, the number of connections is represented by a policy variable, and the PDP has to retrieve the value of this variable to decide whether a new connection can be opened, and to increase the variable value to represent the fact that a new network connection has been opened. In these cases the PDP interacts with the Policy Information Service (PIS). The PIS is a further component of the S3MS framework architecture that is in charged of managing the state of the policy. The PDP could also need some information about the current state of the device to evaluate the policy. As an example, a policy could state that an SMS message can be sent only if the battery level is above a given threshold. In this case, the PDP interacts to a further component of the architecture the System Information Service (SIS). In particular, the following information can be requested to the SIS: get date and time, get CPU load, get free memory size, get network type, get battery level.

6 S×C as a Service Oriented Architecture

This section presents the service-oriented architecture of the S×C framework in the spirit of the SECSE conceptual model. The architecture consists of several layers, defines the S×C services provided by the S×C framework, and has been designed with the following goals in mind:

- The application/service life-cycle (as shown in Fig. 1) is quite complex and a number of stake-holders are involved; at least two: developer and user. The developer is the subject that wrote the application code, while the user is the mobile device owner that wants to execute the application. Another stake-holder is the application provider, who distributes the applications to the users. However, the stake-holders number is often bigger when some framework tasks are outsourced to third parties. Obviously, the various phases of the development, deployment and execution cycle require that data elements are to be exchanged between stake-holders. Consequently, one set of functions of the S×C architecture is concerned with protection of the communication and the exchanged data elements between stake-holders.

- In a real business model some support for accounting, charging and billing needs to be provided which requires that service usages, e.g. downloading the client.

- For a certification process trust relationships between stake-holders need to be established. We do not require that this is done explicitly, but we rely on a certification infrastructure that manages public key certificates for digital signatures. The word "trust" here refers to the authenticity and integrity of exchanged data elements such as code, contract, proof and policy. Specifically, authenticity and integrity of data elements becomes important when one thinks about that in-lining of code (i.e. code is added to the byte-code of a application) for contract and policy compliance is done by a third party.

In order to define the architecture we have carried a careful threat analysis which is reported in the S^3MS deliverables and is synthetised in Fig. 6. It combines parties and data elements and identifies possible threats.

Code, contracts and policy are maintained in the domain of the application, service developer, operator, and therefore the protection that they need is guaranteed by the S×C framework itself. Let's consider the case of static analysis:

By the developer. In that case, the developer uses the static analysis tool during the development process in order to verify that the application matches its contract. A

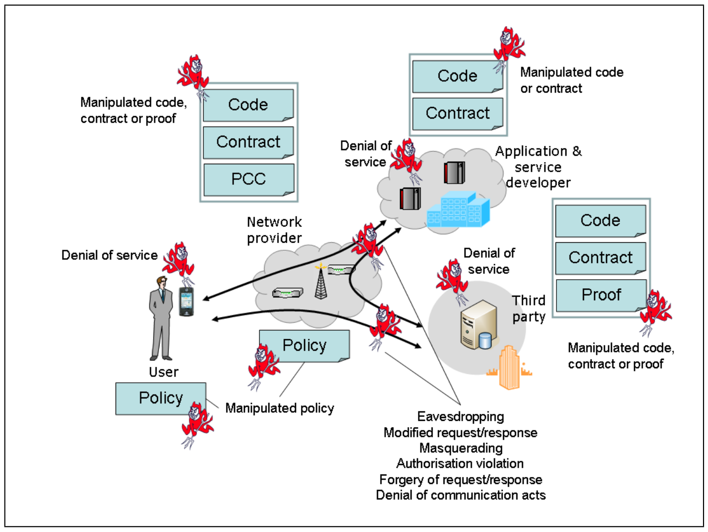

Manipulated code, contract or proof
Code
Contract
PCC
Code
Contract
Manipulated code or contract
Denial of service
Application & service developer
Code
Contract
Proof
Network provider
Denial of service
Denial of service
Third party
Manipulated code, contract or proof
User
Policy
Policy
Manipulated policy
Eavesdropping
Modified request/response
Masquerading
Authorisation violation
Forgery of request/response
Denial of communication acts

Figure 6: Threat Analysis

successful result, in which the static analysis tool is able to prove that the application actually complies with its contract, can then be used by the developer as a support to claims about the innocuity of the application.

By the application provider. The application provider acts as a producer and distributor of the application, pushing it for deployment by network and portal operators. After reviewing the application's contract and ensuring that it is compatible with typical policies, the application provider will verify as part of the quality assurance process that the application actually satisfies its contract, before offering it for deployment.

By the application portal/network operator. The operator in charge of the application's deployment (whether it is an application portal operator or a network operator) usually has contractual obligations regarding the content that it proposes for downloading. It will therefore verify that an application's contract matches its policies, and that the corresponding application matches its application contract.

Another example, the proof of compliance may be given as PCC (proof-carrying-code, see [18]) in which case it is maintained in the domain of the developer. Policies are a concern of the user and the mobile network operator. In the S^3MS project KTH has developed a PCC for Java that is currently visible on the web site of the project among the deliverables.

The analysis of the threats in the communication between the stake-holders during the application life-cycle showed a number of low-level security issues that do not rely on the S×C feature and therefore must be secured independently: eavesdropping, modifying request or response, masquerading, forgery of request or response, authorization violation. The first four threats obviously impact the overall security of S×C as no guarantees on the authenticity and integrity of S×C components and policy can be given without providing some security mechanisms for S×C .

DoS attacks can be run against any of the parties and may block them from providing services. As an example, running a DoS attack against a party that performs the contract and policy matching, prevents the users that need this service to execute their applications. Taking the threat analysis and goals from above in account and following the definition of a security architecture given in [22], a layered S×C architecture with a strict separation between and allocation of services to the layers is motivated. Our proposed S×C architecture differentiates four layers as depicted in Fig. 7.

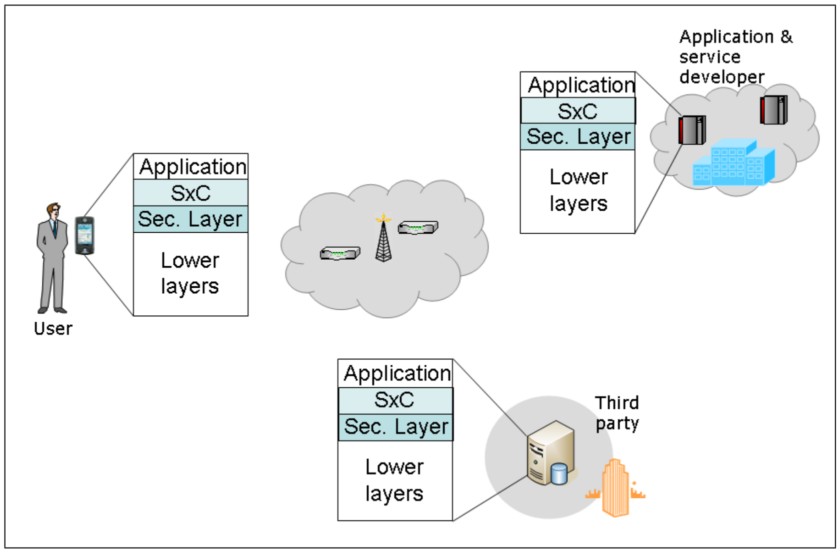


Figure 7: S×C Architecture

The Service/Application Layer defines the top layer of the architecture, and rests on the S×C layer. The services being provided depend on the usage scenario of the service layer:

- The user experiences the application layer as the layer where the downloaded mobile applications are being executed.
- For the developer the layer integrates the development tools that are being used for the implementation of the mobile application.
- Lastly, the third party runs its certification process in this layer.

The S×C services, being part of the S×C layer, are detailed on the left column of Table 2. These services are invoked by the user to download code (from the developer or a third party) and to initiate specific S×C services. The services map to the enforcement methods that are applicable along the softwar life-cycle. To run an application on a mobile device, a user may first get the application code and the contract, analyse code and contract to check their compliance, performs the matching of contract against the platform policy and then, if all previous steps were successful, executes the downloaded application. Eventually, the execution may be monitored. Similar interaction patterns can be derived for the other S×C services.

Table 2: S×C services

Service	**Local**	**Remote**
get(Code, Contract)	on-line	on-line
analyze(Code, Contract)	N/A	off-line
inline(Code, Contract)	N/A	off-line
inline(Code, Policy)	on-device	on-line/off-line
match(Contract, Policy)	on-device	on-line
monitor(Code, Policy)	on-device	N/A
check(Code, Contract, Proof)	on-device	on-line
proofGen(Code, Contract)	N/A	off-line
manage(Policy)	on-device	on-line/on device

get(Code, Contract) This service returns the requested code and contract from either the developer or a third party.

analyze(Code, Contract) Code and contract are analysed for compliance and, if successful a positive result is returned to the caller; and a negative result otherwise.

inline(Code, Contract) The code is submitted to the in-lining service for code and contract compliance assurance. This service returns the in-lined code, i.e. code that has been rewritten in order to embed a monitor directly in the code itself.

inline(Code, Policy) The code is submitted to the in-lining service for code and policy compliance assurance. This service returns the in-lined code.

match(Contract, Policy) Contract and policy are analysed and the compliance of contract to policy is checked. If compliance can be established a positive, otherwise a negative result is returned.

monitor(Code, Policy) The monitoring of the code with respect to the policy is initiated. The monitoring terminates if a policy violation is being detected.

check(Code, Contract, Proof) The proof is checked against the given code and contract. If the check is successful a positive result is returned; otherwise a negative result is returned.

prove(Code, Contract) The compliance of code and contract is established and a respective proof is returned.

manage(Policy) The service enables a party to create, update or delete its own policies. An updated policy is returned (in case of the deletion of a policy nothing is returned).

Depending on the complexity of the code, its contract and the policies to be obeyed, execution of some of the S×C services is demanding with respect to computational power and memory capacity of the executing devices. Considering mobile applications being executed on current constrained mobile devices, it is obvious that some services cannot be executed on these platforms. This requires to outsource respective services. In Table 2 we consider two interesting cases from the business model perspective of S×C for pervasive services, which also take into account the relative computational complexity of each of the tasks. In the *local services* model the device is powerful enough to perform most tasks by itself or rely on the developer to check a number of them. In the *remote services* model (value-added service model in the terminology of mobile operators) the S×C services are offered by a third party such as the mobile operator of the nomadic device.

- The developer can use proof-carrying-code [19] for the proof of compliance of code and contract. Later on, the user checks the correctness of the proof-carrying-code on his device by calling the respective *check* S×C service. This proof checking is restricted in complexity and can be done on the user's mobile device.
- Code and contract compliance by in-lining is a service that should be carried out by a third party only. Done by the developer, the user does not get any evidence that the in-lining has been performed correctly. The third party on the other hand, as part of the certification process asserts that the in-lining is correct and covers all of the properties of the contract.
- Lastly, monitoring execution of the mobile application code for policy compliance is reasonably to be done on the mobile device only.

The second and third columns of Table 2 extend this discussion to all S×C services. Further, with *on-line* we refer to a service that might be executed on-demand, i.e. a communication with the pervasive environment is established in order to set up a channel between the developer's, third party's or user's devices. A service is offered *off-line* if a service is requested and executed in advance. This is an option for the *in-lining* of code for contract compliance which can be done much in advance of the actual mobile application code download.

Security Layer Whenever S×C services require the cooperation of an external party, this holds for all the S×C service marked *on-line* or *off-line* in Table 2, the S×C layer calls the services of the security layer to protect the communication between the parties.

The interactions between the described layers are as follows. Assuming that the user of the mobile device is performing a download of an application from the application and service developer then the user calls the respective method of the S×C layer (i.e. get(Code) from (Developer)). Further, this call is mapped into a respective call to the security layer. The security layer subsequently maps this call into a sequence of method calls to set up a connection with the developer, performs authentication of the developer, gets the package containing code, contract and signature, checks the signature for correctness, extracts code and contract from the received package and hands code and contract back to the calling S×C layer which in turn gives the code and contract back to the application layer.

Lower layers These simply support the security layer in its communication with other mobile devices and servers. We assume the lower layers are comprised of a TCP/IP protocol stack.

7 Conclusions

The S^3MS project belongs to the area of Service Oriented Computing, and address issues, the security and trust, that cut across all the three research planes of the SOC roadmap (Foundation, Composition and Management and Monitoring) [6]. As a matter of fact, the S3MS project designed and developed a contract-based approach to develop secure service in the mobile device scenario. The project concerns at least the three characteristics that the SOC roadmap defines for transversal services (Semantics, Non functional characteristics and Quality of Services). As a matter of fact, the S3MS project addresses non functional characteristics of services, such security and trust in the mobile device scenario, where contracts define the semantics of the applications from the security point of view. From the point of view of the Quality of Service (QoS), in the case of the S3MS project we can talk of Quality of Protection (QoP), since the quality attribute that is guarantee by the S×C framework is security. Hence, all the stake-holders of the S3MS scenario exploit this approach, i.e. the services provided by the S×C framework, to develop, deploy and execute new secure services for mobile devices. As an example, in the S×C model, application developers are responsible for delivering a security contract together with each application. Contract are written using the language defined by the S×C framework, and will be managed by the other stake-holders of the scenario with the services provided by the the S×C framework too.

Here we have proposed a framework and a technological solution for trusted deployment and execution of communicating mobile applications in heterogeneous environments where a *contract-based security mechanism* lies at the core of the framework. In S×C a contract is a claim by a mobile application on the interaction with relevant security and privacy features of a mobile platform. Here we have shown how the compliance of the contract with the application can be verified, how contracts could be matched by policies during deployment and how can they be enforced during development, at time of delivery and loading, and during execution of the application by the mobile platform.

We argue that in the long term new paradigm will not replace, but enhance today's security mechanism, and will provide a flexible, simple and scalable security and privacy protection mechanism for future mobile systems. It will allow a network operator and a user to decide what an application is allowed to do, prevent bad code from running, and allow good code to be easily designed and deployed.

In this way we hope to build the basis for a concrete opening of the software market of nomadic devices to trusted third party applications, without sandboxing and without the burden of roaming trust infrastructure.

References

[1] I. Aktug and K. Naliuka. Conspec - a formal language for policy specification. To appear in Proc. of The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007), 2007.

[2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *CCS '99: Proceedings of the 6th ACM conference on Computer and communications security*, pages 52–62, New York, NY, USA, 1999. ACM Press.

[3] J. Bacon. Toward Pervasive Computing. *IEEE Perv.*, 1(2):84–86, 2002.

[4] Building bug-free O-O software: An introduction to Design by Contract. Eiffel manual available at http://archive.eiffel.com/doc/manuals/technology/contract.

[5] D. Chakraborty, K. Dasgupta, S. Mittal, A. Misra, A. Gupta, E. Newmark, and C. Oberle. Businessfinder: harnessing presence to enable live yellow pages for small, medium and micro mobile businesses. *IEEE Comm.*, 45(1):144–151, Jan. 2007.

[6] E. di Nitto, P. Traverso, A. Sassen, and A. Zwegers. At your service: An overview of results of projects in the field of service engineering of the IST programme. Book introduction, 2007.

[7] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. *IEEE Network*, 13(4):6–15, 1999.

[8] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *EuroPKI 2007: Proceedings of the Fourth European PKI Workshop: Theory and Practice*, pages 297–312. Springer-Verlag, 2007.

[9] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, page 0246, New York, NY, USA, 2000. IEEE Computer Society.

[10] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation.* Pearson Education, 2003.

[11] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The Anatomy of a Context-Aware Application. *WiNet*, 8(2 - 3):187–197, 2002.

[12] JSR 118 Expert Group. Mobile Information Device Profile for Java 2 Micro Edition. Java Community Process, JSP 118, 2002. Available at http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html.

[13] JSR 118 Expert Group. Security for GSM/UMTS Compliant Devices Recommended Practice. Addendum to the Mobile Information Device Profile. Java Community Process, 2002. Available at http://www.jcp.org/aboutJava/communityprocess/maintenance/jsr118/.

[14] B. LaMacchia and S. Lange. *.NET Framework security.* Addison Wesley, 2002.

[15] F. Martinelli, P. Mori, and A. Vaccarelli. Towards continuous usage control on grid computational services. In *In Proceedings of International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services 2005.* IEEE Computer Society, 2005.

[16] Framework- and Application-Specific Security Requirements. MOBIUS (Mobility, Ubiquity and Security) deliverable D1.2, October 2006, http://mobius.inria.fr/twiki/pub/DeliverablesList/WebHome/Deliv1-2corrected.pdf.

[17] F. Montagut and R. Molva. Enabling Pervasive Execution of Workflows. In *Proc. of 1st IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 2005.

[18] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[19] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Mobile Agents and Security*, pages 61–91. Springer-Verlag, London, UK, 1998.

[20] M. Nordio, R. Medel, F. Bavera, J. Aguirre, and G. Baum. A Framework for Execution of Secure Mobile Code based on Static Analysis. In *QEST '04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference on (QEST'04)*, pages 59–66, Washington, DC, USA, 2004. IEEE Computer Society.

[21] N. Paul and D. Evans. .NET Security: Lessons Learned and Missed from Java. In *Proc. of ACSAC'04*, 2004.

[22] R. Shirey. Internet Security Glossary, 2000. RFC 2828.

[23] C. Skalka and S. Smith. Static enforcement of security with types. *ACM SIGPLAN Notices*, 35(9):34–45, 2000.

[24] Sun Microsystems Inc. The Connected Limited Device Configuration Specification. . Java Community Process, JSR 139, 2003. Available at http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html.