What the Heck is this Application doing? - A Security-by-Contract Architecture for Pervasive Services*

N. Dragoni
Technical University of Denmark
ndra@imm.dtu.dk

F. Massacci
University of Trento
massacci@dit.unitn.it

T. Walter and C. Schaefer
DOCOMO Euro-Labs
surname@docomolab-euro.com

Abstract

Future pervasive environments are characterized by non-fixed architectures made of users and ubiquitous computers. They will be shaped by *pervasive client downloads*, i.e. new (untrusted) applications will be dynamically downloaded to make a better use of the computational power available in the ubiquitous computing environment.

To address the challenges of this paradigm we propose the notion of *security-by-contract* (S×C), as in programming-by-contract, based on the notion of a mobile contract that a pervasive download carries with itself. It describes the relevant security features of the application and the relevant security interactions with its computing environment. The contract can be used to check it against the device policy for compliance.

In this paper we describe the S×C concepts, the S×C architecture and implementation and sketch some interaction modalities of the S×C paradigm.

Keywords: security-by-contracts, security architecture, pervasive downloads, pervasive services, policies, policy enforcement.

*This work is partly supported by the project EU-IST-STREP-S3MS (www.s3ms.org). A preliminary, much shorter version of this paper has been accepted to IEEE SecPerU-07.

What the Heck is this Application doing? - A Security-by-Contract Architecture for Pervasive Services*

Abstract

Future pervasive environments are characterized by non-fixed architectures made of users and ubiquitous computers. They will be shaped by *pervasive client downloads*, i.e. new (untrusted) applications will be dynamically downloaded to make a better use of the computational power available in the ubiquitous computing environment.

To address the challenges of this paradigm we propose the notion of *security-by-contract* (S×C), as in programming-by-contract, based on the notion of a mobile contract that a pervasive download carries with itself. It describes the relevant security features of the application and the relevant security interactions with its computing environment. The contract can be used to check it against the device policy for compliance.

In this paper we describe the S×C concepts, the S×C architecture and implementation and sketch some interaction modalities of the S×C paradigm.

Keywords: security-by-contracts, security architecture, pervasive downloads, pervasive services, policies, policy enforcement.

1 Introduction

Security and trust have been identified as key issues in the pervasive computing vision from their earliest inception [1]. Indeed, the paradigm of pervasive services [2] envisions a nomadic user traversing a variety of environments and seamlessly and constantly receiving services from other portables, hand-helds, embedded or wearable computers. Bootstrapping and managing security of services in this scenario is a major challenge as downloaded code might be malware using too much resources of the device or even harm the device.

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

*This work is partly supported by the project EU-IST-STREP-S3MS (www.s3ms.org). A preliminary, much shorter version of this paper has been accepted to IEEE SecPerU-07.

the smart phone in 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 [3]. Even when it is decentralized to increase scalability and performance [4, 5], such distribution does not exploit the device's 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-services-like fashion (either in push or pull mode) but also download *new* applications that are able to exploit the computational power of the user's device. For instance, in order to make a better use of the services available in the environment.

Client downloads create new threats and security risks on top of the "simple" pervasive service invocation because it violates the model of mobile software download behind the Java [6] and .NET mobile security architectures [7, 8]:

- Most pervasive software producers are small and medium sized enterprises (SME) which cannot afford the costs of certification necessary to obtain an operator's certification and thus the downloaded application will not run as trusted code.
- A pervasive download is essentially untrusted code whose security properties we cannot check and whose code signature (if any) has no degree of trust.
- According to the classical security model it should be sandboxed, its interaction with the environment and the device's 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 is trustworthy, i.e. not harming the host system, being developed to exploit the business opportunities of pervasive services.

Contributions of the Paper. Given the above considerations, our contributions are as follows:

- We develop the concept of *Security-by-Contract* (S×C) as a mechanism to make the trust-less download of code possible. S×C covers all stages of the software life-cycle: from design and development to execution. The key idea behind S×C is that the result of each stage of the software life-cycle is verified against defined properties and, if verification was successful, it is forwarded to the next stage. Besides generic hardware and software platform properties, verification may as well take policies into account. Mobile users have an interest that downloaded code respects their policies, e.g., which

communication resources can be used to what extent so that malware is prevented from using too many resources. Mobile operators have an interest that the application does not harm the functioning of the mobile device. S×C enables the trustworthy (modulo above mentioned properties and policies) execution of the downloaded code on the user's mobile device.

- Although designed to cover the software life-cycle, S×C provides the flexibility to choose among the tools, to skip verifications and to enter the process at any point of the life-cycle. Where to enter the workflow depends on the available data.
- Although the computational power of mobile devices is steadily increasing, it may not be sufficient to perform some of the verification steps on the device itself. To cope with this situation, the S×C paradigm allows for an outsourcing of some verifications to (trusted) third parties. Involving third parties, however, requires that the communication between the involved parties (or stakeholders) is being protected. Thus, the S×C concept has to be embedded into an architecture that provides access to the S×C services for performing mentioned verification and which is supported by a security service.

Outline of the Paper. We start discussing related work (§2) to motivate the need for a generic security framework for pervasive services. Then we describe the Security-by-Contract S×C paradigm in detail (§3) and discuss the phases of the software life-cycle and applicable verification techniques. Further, we discuss our layered security architecture and security services (§4) supporting the S×C paradigm, and discuss the vulnerabilities and mitigation strategies of the employed security services (§5). We highlight our implementation of the S×C architecture (§6) and sketch some interaction modalities of the S×C paradigm (§7) before we conclude.

2 Related Work

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

Sandbox Security Model. This is the original security model provided by Java. The essence of the approach [9] 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 easily

recognizable: it can provide security 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 [6], 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 runtime *stack inspection* technique [10].

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 runtime. 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, the 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. [11] address the software aging problem proposing a method that employs an executable content loader and a short-lived configuration management file.

Certification systems like "Symbian signed"[1] are only guaranteeing that, for example, the identity of the application provider has been checked and that the application installs files in specific directories. Technically speaking, the certificate identifies the application's origin, and grants access to those capability-protected APIs that the application declared at build-time. But this approach has two main limitations. Firstly, it is worth noting that capabilities protecting the most sensitive system services (such as Trusted Computing Base or All Files, that is the so-called "Device Manufacturer Capabilities" in the Symbian OS jargon) are not granted for all Symbian signed applications. These capabilities are only available through the "Open Signed" (with a Publisher ID) and "Certified Signed" options. Moreover, as

[1] https://www.symbiansigned.com/

part of the signing process, an application requiring any of these capabilities must gain the approval of a phone manufacturer. Secondly, in the signing process it is not checked if the application is doing things not wanted by the user e.g., sending data with information the user does not want to be sent (see F-Secure Weblog (www.f-secure.com/weblog) from May 11th 2007, "Just because it's Signed doesn't mean it isn't spying on you").

Proof-Carrying Code (PCC). The PCC approach [12] 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 [13] by means of a static analysis of the producer code. The PCC approach is problematic for two main reasons [14]. 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 (MCC). 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 [15]. 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.

Many attempts have been made to apply the above traditional security concepts and solutions to pervasive platforms. However, in most cases, a lot of modifications are needed in order for the security infrastructure to fit within the pervasive framework leading to a high level of risk of introducing new breaches [16]. The infrastructure supporting a pervasive computing system introduces new security challenges not addressed in existing security models, including in the domain of trust management [16, 17, 18]. As a result, a generic security framework for pervasive services is needed [19].

3 Security-By-Contract Framework

In this section we present the overall S×C framework proceeding as follows. First, we focus on the S×C innovative components (§3.1), namely *security contract* and *policy*. Then, in §3.2, we describe the phases of the mobile application life-cycle in which the S×C paradigm is present.

3.1 S×C Innovative Components

The environment is shaped by four groups of stakeholders: *mobile users, service providers or developers, mobile operators, and third parties*, shown in Fig. 1.

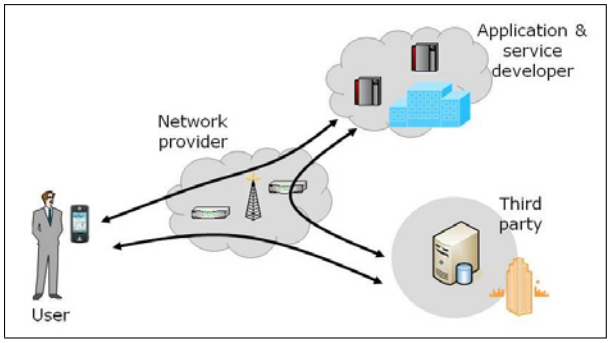


Figure 1: S×C stakeholders

Mobile code developers provide a description of the security behavior that their code exhibits.

Definition 1 (Contract) *A contract is a behavioral specification of the security relevant actions (VM API Calls, Operating System Calls) of the application.*

Loosely speaking, a *contract* contains a description of the relevant features of the application and the relevant interactions with its host platform. Security contracts may include fine-grained resource control (e.g., silently initiate a phone call or send an SMS), memory usage, secure and insecure web connections, user privacy protection, confidentiality of application data and constraints on access from other applications already on the platform.

Example 1 *Examples of security contracts for mobile applications include:*

- *The application sends no more than a specified number of messages in each session.*
- *The application only loads each image from the network once.*
- *The application does not initiate calls to international numbers.*

- *The application does not send MMS messages.*
- *The application connects only to its origin domain.*
- *The application does not use the* FileConnection.delete() *function.*
- *The application only receives messages on a specific port.*
- *A message sent does not exceed the payload of a single SMS message.*
- *The application closes all files, it had opened.*

On the other hand, mobile users and mobile operators would like that any downloaded application respects their policies. And, these policies should be matched by the application's contract.

Definition 2 (Policy) *A policy is a specification of the acceptable behavior of applications to be executed on the platform with respect to security actions.*

Permissions can be granted or not, depending, for instance, on previous actions of the application or some conditions in the application environment. Let us look at two examples.

Example 2 *Personal information security and preventing a device to run out of battery using wireless connections can be ensured by the following policies:*

- "No external connections are allowed if the application has accessed the user's personal information." *Here granting the permission depends on the application's previous actions.*
- "The application is not allowed to use wireless connection if the battery level is below a certain limit". *Here granting the permission depends on the state of application's environment.*

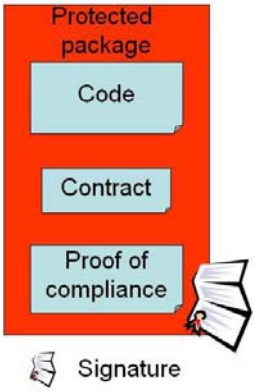
With the security-by-contract paradigm each "application" may consist of four components (Figure 2):

Executable Code It may well be native code, though we focus on managed code and notably CLI and Java byte-code based solutions.

Runtime level contract The component just introduced.

Proof of compliance An optional component that allows for a quick verification that the code actually complies with the claims stated in the contract (aka proof-carrying-code).

Application credentials Credentials (signatures, certificates, etc.) needed by the application to run.



Executable code Managed or native code for the nomadic device.

Runtime level contract Component describing all security relevant actions and behavior of executable code.

Proof of compliance Optional component supporting an efficient verification that the code complies with claims stated in the contract.

Application credentials (Signatures, certificates, etc.) needed by the application to run.

Figure 2: Components of a S3MS Application

By signing the code the developer binds it with the claims on its security-relevant behavior, i.e. its contract, and thus provides a *semantics to digital signatures*. This represents one of the key ideas behind the S×C approach: a digital signature should not just certify the origin of the code *but rather bind together the code with a contract describing its security relevant features*.

3.2 S×C Life-Cycle

Figure 3 summarizes the phases of the mobile application life-cycle in which the S×C paradigm is present. In order to guarantee that an application complies with its contract or the user's policy we should consider the stages where such enforcement can be done, as shown in Tab. 1.

Table 1: Enforcing S×C at Different Stages

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

Enforcing at level (I) can be achieved by appropriate design rules and requires developer support; (II) and (III) can be carried out through (automatic) verification techniques. Such verifications can take place before downloading (e.g., *static verification* [20, 21] done to prove compliance of code and contract) or as a combination of pre and post-loading operations (e.g., through *proof carrying code* [22] and *in-line monitors* [23, 24]); (IV) can be implemented by *runtime checking* [25, 24].

With reference to Fig. 3 an idealized workflow runs as follows. The first step is the *contract authoring* stage in which a contract for the application is specified.

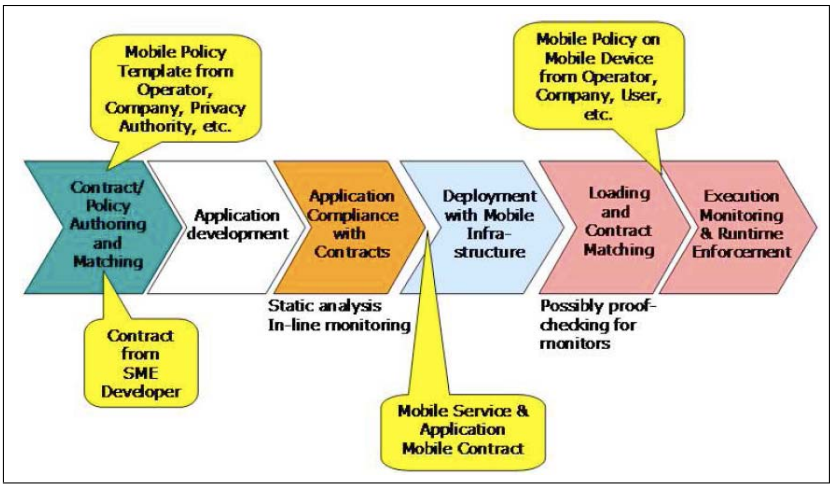


Figure 3: Mobile Application Life-Cycle

The contract is used to express requirements to the application development team, requirements that are being derived from contracts of already existing applications, or from a template of the mobile operator. Considering the requirements from the mobile operators already during development eases deployment of the mobile application later on.

Once we have the right contract and application at hand, we have the problem of *compliance of the application with the contract*. The proof of compliance can be generated by running a static analysis on the code and check for the properties as claimed in the contract. Alternatively, the code might be in-lined so that the claims of the contract are enforced by the in-lined code. If either verification methods succeeds the workflow may continue.

The next step then is of *contract and policy matching*. This represents the basic tenet of the S×C vision: by augmenting applications with contracts we can decide *before* running an application on our mobile whether its security behavior is compatible with our security policy. Specifically, contract and policy matching is done by proofing that the requirements as defined by the policies imply the properties of the application as defined by the contract. For our S×C paradigm, the approach is to map contracts and policies to sets of traces and checking for trace set inclusion [26] or simulation [27].

The last and final step addresses what happens if we "really" want to run an application, even if matching failed. A solution is runtime monitoring where the application is executed under the control of a monitor. The monitor intercepts all security relevant actions and checks them against the applicable policy. Execution of the application is terminated if the policy is violated.

Besides the described workflow, other workflows are conceivable. As depicted in Fig. 4, another workflow starts with the enforcement of the policy either by in-

lining or runtime monitoring. The latter is always an option and applicable even if no contract for the application is available.

Note that only contract and policy matching is not sufficient neither does only code and contract verification suffice. The combination of technologies along the complete mobile application life-cycle gives the guarantee that the downloaded code complies to its contract and obeys defined policies.

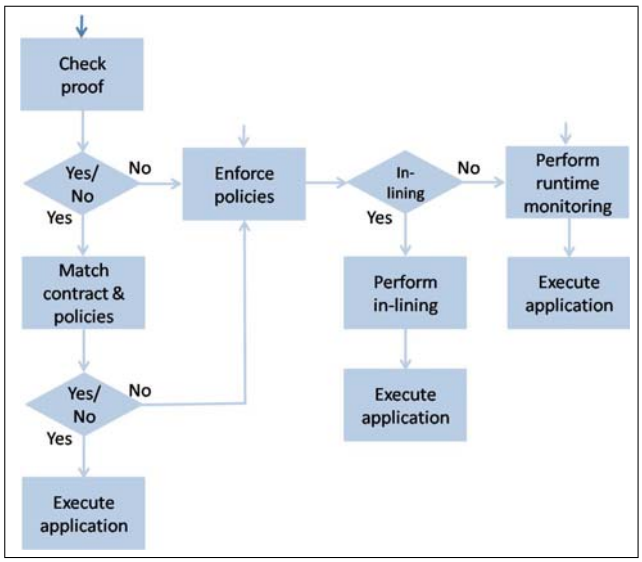


Figure 4: Enforcement strategies workflow

Tab. 2 shows strengths ($\surd$) and limitations ($\times$) of each technology w.r.t. the identified requirements.

4 Layered S×C Architecture

The S×C architecture, to be further discussed below, supports the mobile application life-cycle (Figure 3) while dealing with the different technical and business impacts of the verification methods.

- The mobile application life-cycle is quite complex and a number of stakeholders are involved: developer and user but eventually more when some tasks are outsourced to third parties. Obviously, the various stages of the development, deployment and execution cycle are triggered by data elements that are to be exchanged between stakeholders. Consequently, one set of functions of the S×C architecture is concerned with the protection of the communication between stakeholders as well as the protection of exchanged data elements such as code, contract, proof and policies.

Table 2: Technologies Strengths and Weaknesses

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

- For a sustainable business model some support for accounting, charging and billing needs to be provided which requires that service usages, e.g., requesting an in-lining service, can be attributed to a specific entity. This requires authentication and non-repudiation services being specified in the S×C architecture.

4.1 Application and S×C Layer

Above considerations motivate a layered S×C architecture. Our proposed S×C architecture differentiates four layers as depicted in Fig. 5.

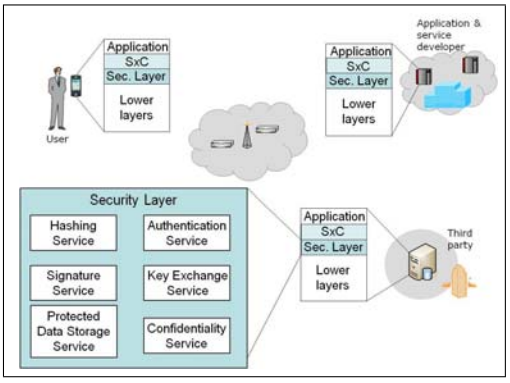


Figure 5: S×C Architecture

The *application layer* defines the top layer of the architecture. The services

being provided depend on the usage scenario:

- The user experiences the application layer as the layer where he can initiate the download of mobile applications as well as initiating the certification process if not already done.
- 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 application layer rests on the S×C layer. The services of the S×C *layer* are listed in Tab. 3. They enable the user to download mobile applications (from the developer or a third party) and to initiate the S×C services.

- The developer can use proof-carrying-code 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.
- Code and contract compliance by in-lining is a service that might be carried out by a trusted third party[2] by invoking the `inline(Code,Contract)` service. The trusted third party may assert that the in-lining is correct and covers all of the properties of the contract. If in-lining is combined with proof-carrying-code then it can be done by the developer himself. The user can verify the result and thus can establish the required trust in the downloaded application.
- Lastly, monitoring execution of the mobile application for policy compliance can reasonably be done on the mobile device only running the `monitor` service.

Depending on the complexity of the code, its contract and the policy, execution of some of the S×C services is demanding with respect to computational power and memory capacity of the executing devices. It is obvious that certain services cannot be executed on these platforms. In Tab. 3 we consider two interesting cases from the point of view of the business models of S×C for pervasive services. Obviously, the more powerful mobile devices become the more S×C services can run on the device. Thus, offering less business opportunities for third parties.

Example 3 *In the* self-service *model the device is powerful enough to perform most tasks by itself* on-device.

Example 4 *In the* value-added services *model the* S×C *services are offered by a third party such as the mobile operator.*

[2] *Trusted third party* is any party that is regarded as trustworthy by the user, e.g., it might be his mobile network operator.

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. For instance, 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 download.

Table 3: Business Model for S×C services

Service	**Behavior**	**Self-Service**	**Value-Added Serv.**
get(Code, Contract)	Gets a specific code and contract from either the developer or a third party.	on-line	on-line
analyze(Code, Contract)	The code is analyzed for compliance with the contract.	N/A	off-line
inline(Code, Contract)	The code is submitted to the inlining service for code/contract compliance assurance.	N/A	off-line
inline(Code, Policy)	The code is submitted to the inlining service for code/policy compliance assurance.	on-device	on-line/off-line
match(Contract, Policy)	The contract is analyzed for compliance with the policy.	on-device	on-line/off-line
monitor(Code, Policy)	The monitoring of the code wrt the policy is initiated.	on-device	N/A
check(Code, Contract, Proof)	The proof is checked against the given code and contract.	on-device	on-line
proofGen(Code, Contract)	The compliance of code and contract is established.	N/A	off-line
manage(Policy)	The service enables a party to create, update or delete policies.	on-device	on-line/on device

We come back to the use of the S×C services when we discuss interaction modalities (§7).

4.2 Security layer

Whenever S×C services require communication with an external party, the S×C layer calls the services of the security layer to protect the communication between the parties. The purpose of the security services is to mitigate vulnerabilities or to detect attacks; some vulnerabilities and attacks are shown in Fig. 6.

The security services offered are listed below. Note that these are comprised of a set of well-known security services except for the protected data service (see below).

Authentication service Authentication mitigates masquerading by respective means, e.g., user name and password, certificates etc. Access control mitigates au-

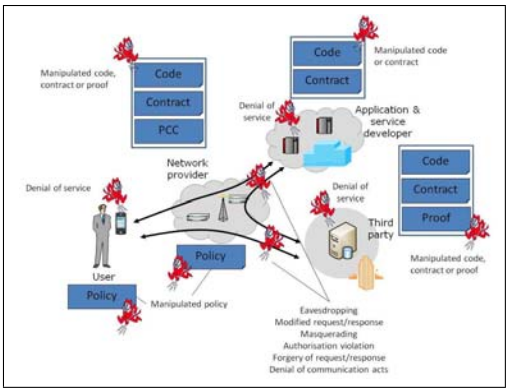


Figure 6: Vulnerabilities and Attacks

thorization violation. Based on the verified identity of the party, it is determined which resources (e.g., code and contract) or services (e.g., in-lining) the party is entitled to use.

Confidentiality service Confidentiality mitigates eavesdropping by encrypting data before sending as only those entities that share the respective keys are able to encrypt and to decrypt the data.

Key exchange service Key exchange is being part of setting up secure communication channels. Depending on the cryptographic algorithm used – symmetric or asymmetric – keys need to be established in advance (symmetric) or are exchanged on demand (asymmetric; using public/private key pairs).

Hashing service Integrity mechanisms such as cryptographic hash functions mitigate the modification of data elements and the modification of request and response messages in the sense that changes are noticeable.

Signing service Signing a data element or a message is a fundamental security mechanism. Besides use cases such as accountability of communication actions as well as non-repudiation, signatures can as well be used to protect data elements such as code and contract. Signing mitigates the threats of modifying code and contract, modifying the message to convey code and contract from developer to user masquerading a developer, forgery of sent messages to contain arbitrary (eventually malicious) code and contract.

Protected data service Obviously, the keys, certificates, policies, etc. are to be stored such that only authorized entities can access them and are protected against tampering. Access to the data service may be protected by a pass phrase, biometric trait, etc.

The interactions between the described layers are as follows (see Section §7 as well): assuming that the user of the mobile device is performing a download of an

application from a developer then the user calls the respective method of the S×C layer (i.e. `get(Code, Contract)`). Further, this call is mapped onto a 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, to perform authentication of the developer, to get the package containing code, contract and signature, to check the signature for correctness, to extract code and contract from the received package and to hand code and contract back to the calling S×C layer which in turn gives the code and contract back to the application layer.

4.3 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.

5 Securing the Mobile Application Life-Cycle

Whereas the S×C framework basically aims at protecting mobile devices against harmful code, the proposed security architecture is aiming at protecting the business processes and software certification process. In this section we look into issues that come up with our S×C paradigm, particularly what may go wrong while running through the stages of the S×C process but as well while stakeholders communicate. The latter refers to securing the communication link between entities utilizing the services of the security layer of our S×C architecture, while the former asks for an assessment of the risks associated with the S×C paradigm.

In the described context but not limited to it, assessing the risks associated with the S×C paradigm means considering which entity is performing the proof (verification) and by what means. As pointed out earlier (Tab. 3) all the S×C services may be executed on-device or on-line/off-line. If done *on-device* then the sole responsibility of establishing the required confidence in the downloaded code is with the device's user. However, the user has to be aware of that the confidence relies on the assumptions that the verification methods are correctly implemented (which we deliberately assume), are correctly executed (i.e. have not been tampered with, which we as well assume as given) and that any verification of code and contract but as well as in-lining and runtime monitoring is done only up to the extent as given by the policies (as defined by user or mobile operator). Note, that the integrity of the systems as well as the S×C services being executed is beyond the scope of this paper. Tamper-resistance techniques may be applicable here.

The previous statements hold as well for the case that certain S×C services are executed on-line/off-line by a trusted third party. By invoking a S×C service remotely, the user is assuming the result – proof of compliance, contract/policy matching, etc. – is correct (modulo the assumption discussed above).

In summary, the security-by-contract paradigm differentiates two levels of trust.

Running the S×C provided verification methods establishes trust in the downloaded application. Utilizing the security services establishes trust in the communication as well in the communication peer.

6 Implementation of the S×C Architecture

Section §4 has defined the security layer and security services of the S×C architecture. Our implementation is utilizing the Secure Socket Layer (SSL) protocol as well as utilizing existing implementations of cryptographic libraries that support the basic cryptographic functions for signing, etc. which we use to implement the required non-repudiation service.

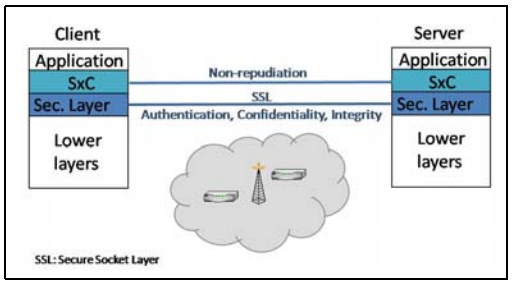
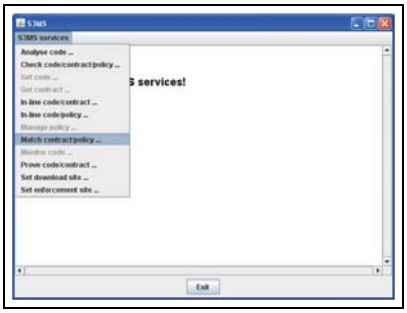


Figure 7: S×C Security Architecture

The implementation of the S×C architecture supports two user interfaces: a graphical user interface (GUI) that runs on S×C clients (Fig. 8(a)) and a portal interface (Fig. 8(b)). Both interfaces provide access to all S×C services (Tab. 3).



(a) Client Graphical User Interface

(b) Portal User Interface

Figure 8: Graphical User Interfaces

As depicted in Fig. 9 the core of the implementation is the server hosting and running the *verification methods*. These are encapsulated as web services which can be invoked by clients using the GUI. Alternatively, a client may get access to

the verification methods via a web portal. The implementation of the S×C architecture provides for a distribution of verification methods across servers that may run and are administered by different domains. As shown in Fig. 9, portal and verification server are in different domains as well is a third server in another domain that runs, for instance, the static analysis verification method. The provided flexibility allows for the support of business models where all verification methods are offered by one administrative domain (which integrates the web services as well as the portal interface) or several domains cooperate, e.g. an mobile network operator (MNO) providing the portal and other third parties the verification methods. For the user, the distribution of services is transparent. The user has to decide which interface to use and which of the S×C services to invoke across domains.

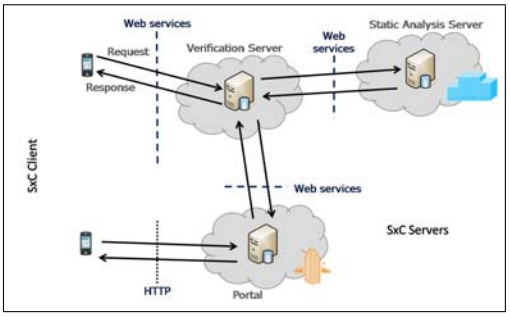


Figure 9: Client, Server and Portal integration

7 Interaction Modalities

As discussed above, the use of verification methods depends on the capabilities of the mobile device that downloads and executes a mobile application. In this section we discuss some usage modalities that represent particularly interesting business cases for value-added service offerings in pervasive environments.

7.1 Policy and contract as well as policy matching on mobile device

The process of downloading mobile applications is initiated by the user. The user has to authenticate with the server which holds the requested code. If required the server authenticates with the user's device. After authentication has taken place, the code is downloaded onto the user's mobile device. Subsequent to the download, the S×C layer (Fig. 5) is taking control of the further steps until execution. Assuming that the downloaded code arrives as in Fig. 2, the following steps are to be done:

1. The signature has to be verified, i.e. the public key of the signing authority has to be retrieved and the signature has to be verified.

2. The contract has to be extracted from the packet containing code as well as contract.
3. The proof of compliance has to be verified.
4. The platform policy has to be retrieved.
5. Contract and platform policy have to be forwarded to the contract-policy matching software module (i.e. operation match(Contract, Policy) is called).
6. If matching is successful the code can be loaded and executed; maybe execution is done under the control of a monitor (i.e. operation monitor(Code, Policy) is called).

7.2 Policy on mobile device and contract and policy matching by third party

With this scenario we build on the previous scenario, i.e. the code is downloaded onto the user's mobile device, but extend the scenario as shown in the sequence diagram of Fig. 10(a). A third party is involved that performs the contract and policy matching. So, the user's device sends the contract and policy to the third party. We assume that the user's device sends a signed message including contract and policy to the third party. Upon reception of the message the third party verifies the message's signature, extracts contract and policy, performs contract-policy matching by calling operation match(Contract, Policy) of the S×C layer, and subsequently returns the signed results back to the user's device.

If all policies are stored with the third party and contract and policy matching is done by the third party, the third party only needs to know which policy it has to match against which contract. This scenario is discussed next.

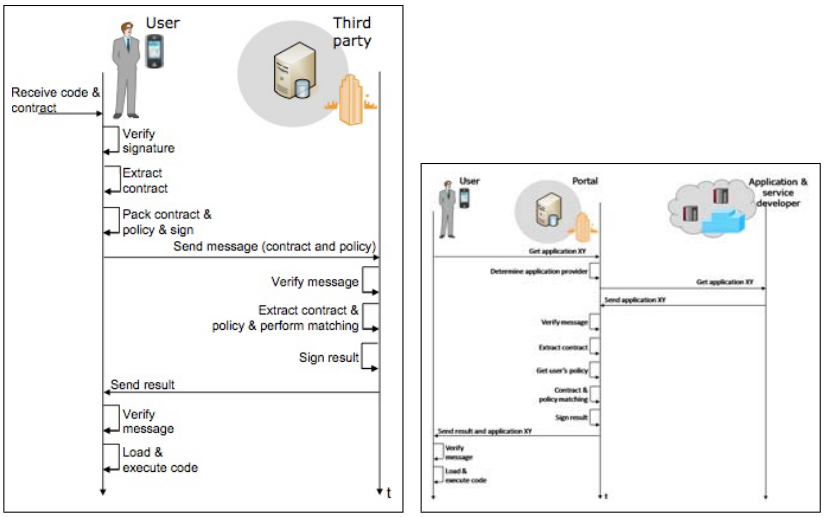
7.3 Policy with third party and contract and policy matching done by third party

The last option being considered is the one where the third party functions as a "portal", i.e. the only interface of the user from his mobile device to the environment is via this portal. The interactions of the user with the portal are shown in Fig. 10(b).

Note that the policy may be defined by the user and transferred to the portal in advance. However, if the portal is run by the user's MNO then it might be very well that the MNO defines a policy for all its users. In this case the policy is already with the portal.

8 Conclusion

In this paper we have proposed the notion of *Security-by-Contract (*S×C*)*, as in programming-by-contract, based on the notion of a mobile contract that a perva-



(a) Policy with user, contract/policy matching by 3rd party

(b) Download and service via portal

Figure 10: Sample Interactions

sive download carries with itself. It describes the relevant security features of the application and the relevant security interactions with its nomadic host. 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.

The main contributions of this paper can be summarized as follows. First, we have described a layered architecture supporting the S×C paradigm for pervasive security. Then we have discussed the threats and mitigation strategies for security services. Third, we have described the implementation of the S×C architecture. Finally, we have sketched some interaction modalities how a user may experience the security-by-contract paradigm.

Acknowledgement

The author would like to thank the anonymous reviewers for their fruitful comments.

References

[1] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

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

[3] 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.

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

[5] D. Chakraborty, K. Dasgupta, S. Mittal, A. Misra, A. Gupta, E. Newmark, and C.L. 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] L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.

[7] Brian LaMacchia and Sebastian Lange. *.NET Framework security*. Addison Wesley, 2002.

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

[9] Li Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, 1997.

[10] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *Symposium on Security and Privacy*, Oakland, CA, USA, 1998. IEEE.

[11] John R. Michener and Tolga Acar. Managing System and Active-Content Integrity. *IEEE Computer*, 33(7):108–110, 2000.

[12] George 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.

[13] George C. Necula and Peter Lee. The Design and Implementation of a Certifying Compiler. *SIGPLAN Not.*, 39(4):612–625, 2004.

[14] 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.

[15] 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.

[16] Michael Collins, Simon Dobson, and Paddy Nixon. Security Issues with Pervasive Computing Frameworks. In *Pervasive'06: Proceedings of the Workshop on Privacy, Trust and Identity Issues for Ambient Intelligence*, pages 679–685. Springer Verlag, 2006.

[17] C. English, P. Nixon, S. Terzis, A. McGettrick, and H. Lowe. Dynamic trust models for ubiquitous computing environments. In *First Workshop on Security in Ubiquitous Computing at the Fourth Annual Conference on Ubiquitous Computing (Ubicomp2002)*, 2002.

[18] C. English, P. Nixon, S. Terzis, A. McGettrick, and H. Lowe. Security models for trusting network appliances. In *5th IEEE International Workshop on Networked Appliances*. IEEE, 2002.

[19] Ghita Kouadri Mostfaoui. Security in Pervasive Environments, What's Next? In Hamid R. Arabnia and Youngsong Mun, editors, *Security and Management*, pages 93–98. CSREA Press, 2003.

[20] X. Leroy. Bytecode verification on java smart cards. *Softw. Pract. Exper.*, 32(4):319–340, 2002.

[21] Gennady Chugunov, Lars-Åke Fredlund, and Dilian Gurov. Model checking of multi-applet javacard applications. In *CARDIS*, pages 87–96, San Jose, CA, USA, 2002. USENIX.

[22] 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.

[23] U. Erlingsson. The inlined reference monitor approach to security policy enforcement, 2003. Technical report 2003-1916, Department of Computer Science, Cornell University.

[24] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *IJIS*, 4(1–2):2–16, February 2005.

[25] Ulfar Erlingsson and Fred B. Schneider. Irm enforcement of java stack inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[26] 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, Mallorca, Spain, 2007. Springer-Verlag.

[27] P. Greci, F. Martinelli, and I. Matteucci. A framework for contract-policy matching based on symbolic simulations for securing mobile device application. In *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 221–236. Springer-Verlag, 2008.

9 Bibliographical Sketch

Nicola Dragoni obtained his M.S. Degree in Computer Science in 2002 and his Ph.D. in Computer Science in 2006, both at University of Bologna. From 2002 to 2006 he also worked as Research Assistant at the Department of Computer Science at the same University. He visited the Knowledge Media Institute at the Open University (UK) and the MIT Center for Collective Intelligence (USA), respectively in 2004 and 2006. In 2007 and 2008 he joined University of Trento as post-doctoral Research Fellow working on the S3MS project. Between 2005 and 2008 he also worked as freelance IT consultant. Since 2009 he is an assistant professor in security and distributed systems at the Denmark Technical University (DTU).

Fabio Massacci received an M.Eng. in 1993 and Ph.D. in Computer Science and Engineering at University of Rome "La Sapienza" in 1998. He joined University of Siena as an assistant professor in 1999, was visiting researcher at IRIT Toulouse in 2000, and joined Trento in 2001, where he is now full professor. His research interests are in security requirements engineering, formal methods, and computer security. His h-index on Google Scholar is 20, and his h-index normalized for individual impact (hI_norm) is 13 (in June 2008). He is currently scientific coordinator of multimillion Euros industry R&D European projects on security and compliance.

Thomas Walter is a senior manager in the Smart and Secure Services Group of DOCOMO Euro-Labs, Germany. His research interests include security of software and services for mobile devices, security policies, and access and usage control in distributed environments. Thomas has an Diploma degree in computer science (University of Hamburg, Germany) and a Doctorate in electrical engineering (Swiss Federal Institute of Technology Zurich, Switzerland). He is a member of Gesellschaft für Informatik (GI) and the IEEE.

Christian Schaefer received his Diploma degree in Computer Science from the University of Karlsruhe (TH), Germany. Since September 2003 he is working as a researcher for DOCOMO Euro-Labs in Munich, Germany. His main research interests are the enforcement of security policies in distributed systems with a focus on usage control and security of mobile handsets. He is a member of IEEE.