

Security-By-Contract (SxC) for Mobile Systems^{*)}

– or how to download software on your mobile without regretting it

NICOLA DRAGONI, FABIO MASSACCI



Nicola Dragoni is an Assistant Professor at Denmark Technical University (DTU)

We present the notion of *Security-by-Contract (SxC)*, a mobile contract that an application carries with itself. The key idea of the framework is that a digital signature should not just certify the origin of the code but rather bind together the code with a contract. 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 platform's security requirements) at deployment time, and its enforcement guaranteed, for instance by in-line monitoring.

We argue that **SxC** would provide semantics for digital signatures on mobile code thus being a step in the transition from trusted code to trustworthy code. **SxC** will not replace, but enhance today's security mechanisms, and will provide a flexible, simple and scalable security mechanism for future mobile systems.



Fabio Massacci is a Professor at University of Trento and a visiting scientist at SINTEF, Norway

1 Introduction

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

One of the reasons for this lack of applications is also the security model adopted for mobile phones. The current security model is exemplified by the JAVA MIDP 2.0 and .NET Compact Framework approach and is based on *trust relationships*: mobile code is run if its origin is trusted. This essentially boils down to *mobile code is accepted if it is digitally signed by a trusted party*. The level of trust of the 'trusted party' determines the privileges of the code by essentially segregating it into appropriate trust domain.

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

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

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

From the point of view of the code producer they produce code with unbounded liability. They cannot declare which security actions the code will do; by signing the code they essentially declare that they did it. The consequence is that injecting an application in the mobile market is a costly operation, as developers must essentially convince the operators that their code will not do anything harmful. As a result, a lot of code comes to the market uncertified or self-certified. In other words, most code is untrusted.

To deal with the untrusted code the mobile version of either .NET [1] or Java [2] enabled devices to exploit the mechanism of permissions. The drawback of permissions is that after assigning a permission the user has very limited control over how the permission is used. An application can receive a permission to send SMS messages and then send hundreds of them invisibly to the user. Once again the consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they are trusted (and then they can do almost everything).

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

^{*)} Research partly supported by the Projects EU-FP6-IST-STREP-S3MS, EU-FP6-IP-SENSORIA, and EU-FP7-IP-MASTER.

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

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

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

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

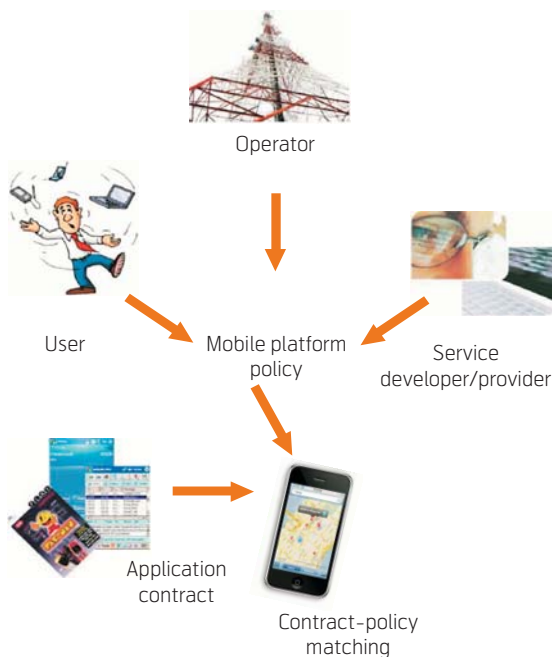


Figure 1 SxC Key Stakeholders

1.1 SxC in a Nutshell

We present in this article the notion of *Security-by-Contract* (as in programming-by-contract [3]): the digital signature should not just certify the origin of the code but rather bind together the code with a contract. Loosely speaking, a *contract* contains a description of the relevant features of the application and the relevant interactions with its host platform. A mobile platform could specify platform contractual requirements, a *policy*¹⁾, which should be matched by the application's contract. Among the relevant features, one can list fine-grained resource control (e.g. silently initiate a phone call or send an SMS), memory usage, secure and insecure web connections, user privacy protection, confidentiality of application data, constraints on access from other applications already on the platform.

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

The article is organized as follows. In the next Section we present the SxC framework providing also a description of the overall lifecycle of mobile code in this setting. In Section 3 we focus on the notions of contract and policy and we briefly introduce their specifications in the framework. In the rest of the article we focus on two technologies successfully deployed in the context of the framework, namely contract-policy matching (Section 4) and inline monitoring (Section 5). Section 6 ends the article with related work and conclusions.

A number of additional results and materials are available on the website of the project at www.s3ms.org.

2 The SxC Framework

The framework of Security-by-Contract for mobile code is essentially shaped by three groups of stakeholders: mobile operator, service provider and/or developer, mobile user. This is shown in Figure 1.

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

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

¹⁾ In the sequel we will refer to policy as the security requirements on the platform side and by contract the security claims made by the mobile code.

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

On the other hand, we can see that users and mobile phone operators are interested in all codes deployed on their platform being secure. In other words, they must declare their security policy:

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

Example 2.1 (Contract & Policy) A user wants to download a SxC compliant chess game. The application comes with a contract consisting of the following two rules:

1. the application only uses HTTP network connections;
2. no messages can be sent by the application.

Figure 2 shows how the user chooses a specific security policy among four pre-set policies. For instance, the user might select a policy consisting of the following two rules:

1. the application uses only high-level (HTTP, HTTPS) network connections;
2. maximum five text messages can be sent by the application.

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

Note the difference with the use of cryptographic signatures in the traditional mobile device security model. In the SxC approach, a signature has clear

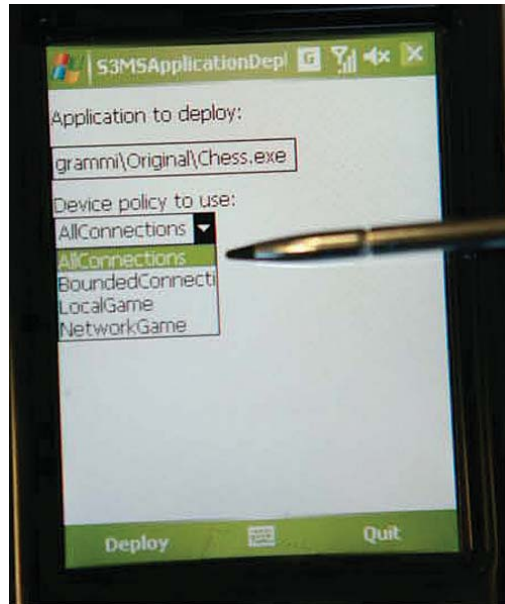


Figure 2 User selects a security policy

semantics: the developer (or third party) claims that the application respects the supplied contract. Moreover, what is important is the fact that the decision whether the contract is acceptable or not remains with the end user.

2.1 The SxC Life Cycle

To take full advantage of this new paradigm, applications have to be developed with SxC in mind. This means that some changes occur in the typical *Develop-Deploy-Run* application life cycle. Figure 3 shows an updated version of the application development life cycle.

The first step to develop an SxC compliant application is to create a contract to which the application will adhere. Remember that the contract represents the security-related behavior of an application. Designing such claim requires intimate knowledge of the inner workings of the application, so it's typically done by a (lead-)developer or technical analyst. Some mobile phone operators, companies or other authori-

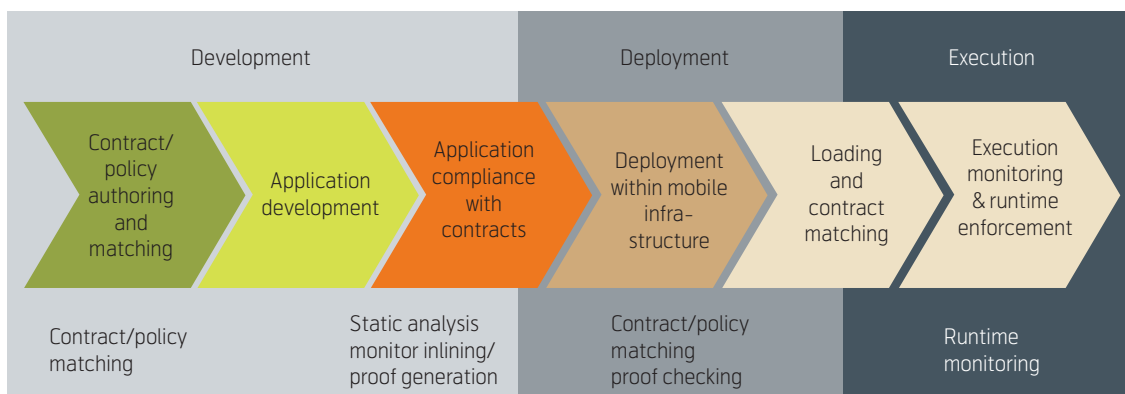


Figure 3 SxC Application/Service Life-Cycle

ties may choose to publish contract templates that can then be used as a basis for new application contracts. Once the initial version of the contract has been specified, the application development can begin. During the development, the contract can be revised and changed when needed.

After the application development, the contract must somehow be linked to the application code in a tamper-proof way. One straightforward method to do this is by having a trusted third party inspect the application source code and the contract. If they can guarantee that the application will not violate the contract, they sign a combined hash of the application and the contract. Another way to link the contract and the code is by generating a formal, verifiable proof that the application complies with the contract, and adding it to the application metadata container. This concept is called *proof-carrying code*. When this step is completed, the application is ready to be deployed. The application is distributed together with its contract and optionally other metadata such as a digital signature from a third party or a proof.

When the program is deployed on a mobile device, the SxC framework checks whether the application contract is compatible with the device policy. This process is called *matching*. What essentially happens is that the SxC framework checks whether the security behavior described in the contract is a subset of the security behavior allowed by the policy. If it is, the application is allowed to run as-is. If the contract is not a subset of the policy, the application is treated as an application without a contract.

2.1.1 Variants of the Development Life Cycle

The scenario in the previous section showed how an application with SxC metadata would be produced and deployed to a mobile device. There is, however, an important need to also support the deployment of applications that are not developed with SxC in mind. This backwards compatibility is a make-or-break feature for the system.

When an application without a contract arrives on the mobile device, there is no possibility to check for policy compliance through matching. No metadata is associated with the application that can prove that it does not break the system policy. A solution for this problem is to enforce the system policy through run time checking.

One example of a run time policy enforcement technology is *inlining*. During the inlining process, the application is modified to intercept and monitor all the calls to the security related events. When the monitor notices that the application is about to break

the policy, the call to the security related event that causes the policy to be broken is canceled.

Alternatively, an explicit run-time monitor integrated for instance in the virtual machine can guard the behavior of the application.

The strength of run time checking is that it can be used to integrate non-SxC aware applications into the SxC process. A result of having this component in the system is that it is usable as-is, without having to update a plethora of existing mobile applications.

A second variant of the development life cycle is where an existing application is made SxC aware. It can sometimes be difficult to compose a contract for an application that may have been written years ago by a number of different developers. Instead of investing a lot of time (and money) into finding out which rules apply to the legacy application, an inlining based alternative could be a solution.

The key idea is to use an inlining technique as described in the first part of this Section. However, instead of inlining the application when it is loaded, the application is inlined by the developer. After being inlined, the application can be certified by a trusted third party.

There are a number of benefits of this approach, compared to on-device inlining. A first advantage is that the parts of the contract that can be manually verified by the trusted third party do not have to be inlined into the application. For instance, imagine that an existing application should comply with the policy “An application cannot access the network, and cannot write more than 1000 bytes to the hard disk.” A third party can easily verify whether the application will break the first part of the policy. If the application contains no network related code, it will not break this part of the policy. The second part of the policy may be harder to verify, if the application does contain some file-related code but if it is unclear how many bytes are actually written to the hard disk. In this case, the developer could inline the application with a monitor that only enforces the second part of the policy, but the application will nevertheless be certified for the full policy.

Inlining large applications on a mobile device can be time consuming. Going through the inlining process before deploying the application eliminates this problem. It speeds up the application loading process, which is a second advantage.

A final advantage is that the developer can do a quality assurance check on the inlined application. This

is useful to weed out subtle bugs and ensure that the inlined application meets the expected quality standard of the developer.

3 What a Contract Should Look Like

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

Further, a number of independent security requirements analyses for mobile and distributed systems [4, 5, 6] show that detailed contracts are not really necessary. Figure 4 shows a distillation of a number of security requirements from these analyses. The characteristic feature of these applications is that they need wide access to services to execute correctly. However, the user still wants to control that these services are not abused or misused. Therefore the same permission can be granted or not granted depending, for instance, on previous actions of the midlet or some conditions on application environment.

Some examples of security policy rules for mobile devices include:

1. The application sends no more than a number of messages in each session.
2. The application only loads each image from the network once.
3. The delay between two periodic invocations of the MIDlet is at least T.
4. The application does not initiate calls to international numbers.
5. The application only uses files whose name matches a given pattern.
6. The application does not send MMS messages.
7. The application connects only to its origin domain.
8. The application does not use the `FileConnection.delete()` function.
9. The application only receives SMS messages on a specific port.

Costly functionality

Any invocation of the paid services, such as sending text messages, using GPRS or wireless connections, must be controllable by the user.

Network connectivity

Any external connections made by the application can be controlled. Conditional network access might be in place.

Information management

It is necessary to control what data is accessed by the application. This includes access to the local files, as well as to PIM items or contacts from Contact List. In consequence of such action different conditional permissions can be assigned.

Interaction with other applets

This requirement makes necessary to control means of interprocess communication, in particular sockets and memory-mapped files. Also starting some midlets might be blocking for other midlets.

Power consumption

This requirement is two-fold: it makes necessary to control the invocation of power-consuming functionality, such as WiFi connections, and to control the battery level while running the application. Conditional to the battery level certain actions might be forbidden.

Extended functionality

If the device is equipped with some advanced functionality, such as camera or GPS receiver, its use is likely to be controlled by policies.

Figure 4 End Users' Distilled Security Requirements

10. The length of an SMS message sent does not exceed the payload of a single SMS message.
11. The application must close all files that it opens.

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

In the context of the SxC framework both contracts and policies can be specified by means of the policy language ConSpec [7] or the logical language 2D-LTL [8]. Figure 5 shows what a ConSpec specification looks like²⁾. Of course, the user is not asked to learn such technical languages for setting a policy for his platform, as the task is surely non-trivial. Indeed, as shown in Figure 2, the user can select the right policy among a pre-set of available policies and using it as-is. Moreover, he can customize a policy by specifying the parameters of the existing policies (e.g. the

2) *Presenting ConSpec is outside the scope of this article. Interested readers are invited to consult [7] for technical details.*

```

MAXINT 10000 MAXLEN 10

RULEID HIGH LEVEL CONNECTIONS
SCOPE Session

SECURITY STATE

BEFORE javax.microedition.io.Connector.open(stringurl)
PERFORM
    url.startsWith("http://") -> {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.TextMessage msg)
PERFORM
    false -> {skip;}

```

Figure 5 ConSpec Specification of the Contract in Ex. 2.1

exact number of the messages that the application is allowed to send). On the other hand, 'informed' users can use a Policy Manager tool to upload their own policies to their devices. But then they must be able to write or to edit the policies in ConSpec or 2D-LTL.

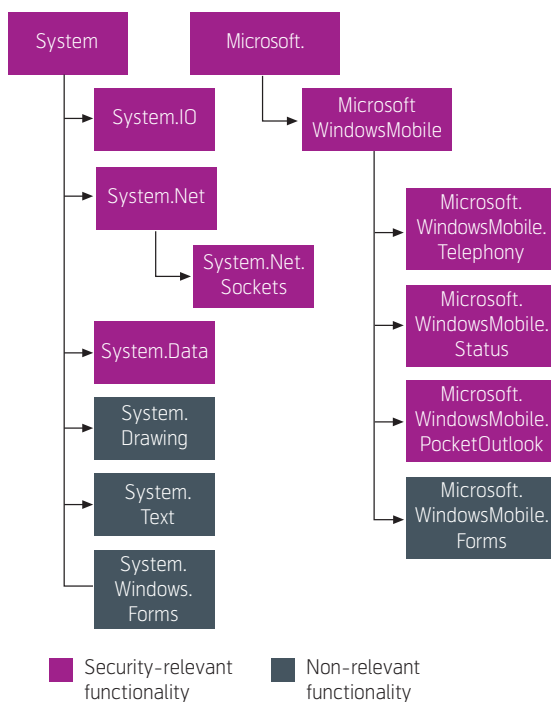


Figure 6 .NET Names and Security Relevant APIs

3.1 Security Relevant or Non-Relevant Actions?

When we deal with concrete mobile applications we need to map the high level requirements into concrete API³⁾ calls. The security policy specified in our language should then guarantee that the security relevant APIs are actually executed in the right order or with the right parameters.

The objective is a tentative classification of the .NET Compact Framework (CF) functionalities into security-relevant and non-relevant. The .NET API is logically organized in the hierarchy of namespaces (see Figure 6), so we use the namespaces as base units of the classification.

While it is obvious that any API call that appears in the policy should be monitored, it is also evident that some functionalities are more likely to be mentioned in security policies than others. We use our distilled security requirements to perform this mapping as shown in Figure 7.

Identifying security relevant actions is harder than it looks and a comprehensive discussion on this issue would be outside the scope of this article. More details can be found consulting the publications and final deliverables on the website of the project (www.s3ms.org). As an exercise for the interested reader we suggest finding the Java MIDP API in the

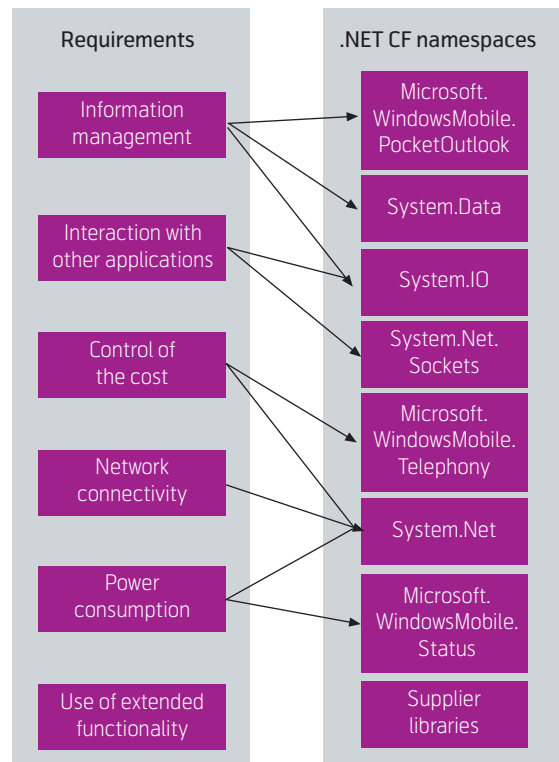


Figure 7 Mapping Requirements to APIs

³⁾ Application Programming Interface

reference implementation that apparently shows a textbox and, by the way, makes a phone call. To this extent .NET is more disciplined than Java [9].

4 Contract-Policy Matching

As sketched in Figure 3, one of the key problems in the overall Security-by-Contract lifecycle is the *contract-policy matching issue*: given a contract that an application carries with itself and a policy that a platform specifies, is the contract compliant with the policy?

Contract-policy matching represents a common problem in the life-cycle because it must be done at all levels: both for development and run-time operation. Intuitively, matching should succeed if and only if by executing the application on the platform every behavior of the application that satisfies its contract also satisfies the platform's policy. To address this issue we have developed efficient algorithms to match application contracts with device policies. Interested readers are invited to consult [10] for the details about the developed theory and algorithms. In Figure 8 we briefly describe the key idea underlying the developed matching algorithm.

Example 4.1 In Example 2.1 we have shown an intuitive example of a contract and a policy that matches. Let us briefly describe how matching works in such situations. In the SxC framework the contract and policy (originally specified in the SxC policy language ConSpec [7]) are translated into automata-based rules [10], that will represent the inputs of the matching algorithm. Figure 9 shows the automata-based specification of the ConSpec specification of Figure 5. Then for each rule in the policy the matching algorithm searches for corresponding rule in the contract and runs an emptiness check algorithm (see Figure 8 for the basic idea). Since the contract allows to use HTTP connections only while the policy allows to use both HTTP and HTTPS connections the obtained result states that the contract matches the policy. For the other rule in the policy an appropriate rule in the contract is found. Here the contract forbids the application to send messages while the policy prescribes that the application can send bounded amount of messages. As a result, the matching algorithm ends successfully: the contract matches the policy. Technically speaking, the matching algorithm has checked whether the security behavior specified in the contract is a subset of the allowed security behavior specified in the policy.

Example 4.2 To see that matching might fail let us consider the following situation. The chess game is now downloaded on a mobile device that has a policy containing the following security rule:

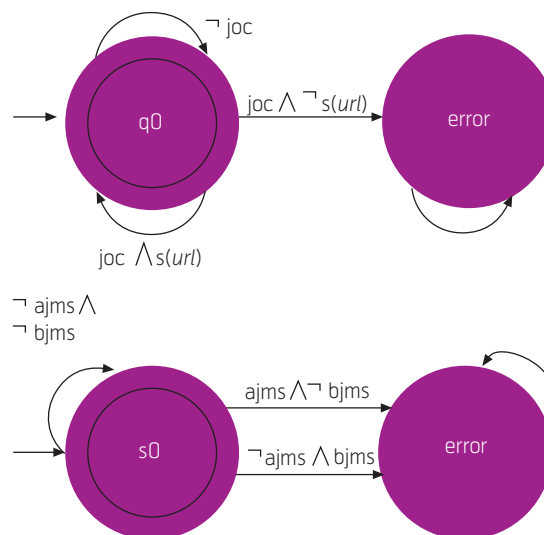
Having contract and policy specified in the SxC policy language ConSpec [7], we would like to concretely solve the problem of matching the security claims of the code (contract) with the security desired by the platform (policy). The problem is solved in [10, 11] introducing the concept of *Automata Modulo Theory (AMT)*, an extension of Büchi Automata (BA) suitable for formalizing systems with finite states but infinite transitions. To represent the security behavior, a system can be represented as an automaton where transitions correspond to the invoked methods as in the works on model-carrying code [12]. In this case, the operation of matching the application's claim with platform policy is a classical problem in automata theory, known as *language inclusion* [13]. Namely, given two automata Aut^C and Aut^P representing respectively the formal specification of a contract and of a policy we have a match when the language \mathcal{L}_{Aut^C} accepted by Aut^C (i.e. the execution traces of the application) is a subset of the language \mathcal{L}_{Aut^P} accepted by Aut^P (i.e. the acceptable traces for the policy). Assuming that the automata are closed under intersection and complementation, the matching problem can be reduced to an emptiness test:

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

Figure 8 Matching as Language Inclusion

- Only urls starting with HTTPS are allowed.

In this case, the contract-policy matching fails. The reason is that the platform allows only secure network connections while the application's contract



Abbreviations for JAVA APIs:

- joc $\hat{=}$ `io.Connector.open(url)`
- $s(url)$ $\hat{=}$ `url.startsWith("http://")`
- $ajms$ $\hat{=}$ `after MessageConnection.send(msg)`
- $bjms$ $\hat{=}$ `before MessageConnection.send(msg)`

Figure 9 Automata for the Contract of Example 2.1

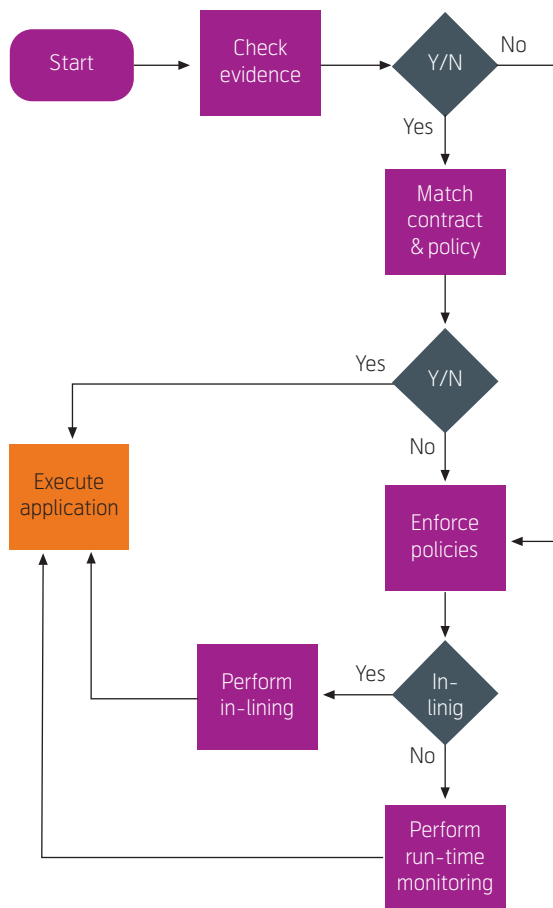


Figure 10 Matching and Inlining

restricts the domain of urls to the ones starting with HTTP only. In other words, the security behavior specified in the contract is not a subset of the allowed security behavior specified in the policy, because there are some actions that are allowed by the contract while forbidden by the policy (for instance, all the urls starting with "HTTP:").

5 Inline Monitoring

As sketched in Figure 10, when matching fails inline monitoring is used to enforce the user policies in a reliable and autonomous way. The goal is to give to the user the opportunity of downloading a completely untrusted program on the device and then to run it safely. For this reason, in the SxC framework we offer the possibility to rewrite the downloaded program inserting (inlining) hooks that notify the monitor before and after each security-relevant event. The inlining is performed directly on the device, so that we do not need to rely on any third party for that. As the result of the inlining each security-relevant method is executed only if it has been allowed by the policy.

Example 5.1 Figure 11 shows two HTC P3600 smartphones playing chess by sending each other SMSs. The first smartphone has no monitoring frame-

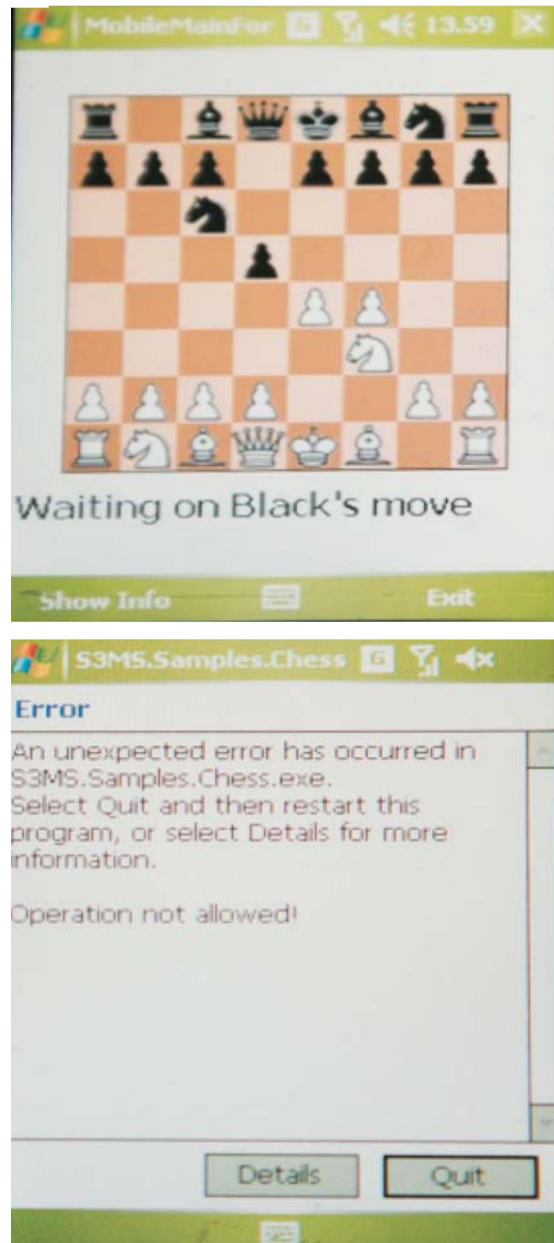


Figure 11 Policy Violation: Error Message

work installed, so the game is allowed to send as many SMSs as it will. The monitoring system of the second phone detected that the user-defined limit of the sent messages was exceeded. For this reason the game is terminated through the security exception.

While the inlining of the downloaded programs is performed on-device, the policies need to be managed off-line. The security policies are written in our policy logical language 2DLTL [8] and then automatically converted in the form suitable for monitoring. This representation of the policy is deployed to the device, where it is used by the inliner to extract the method calls that are relevant to the policy and by the monitor itself.

Summing up, Figure 12 shows our basic usage model that is further described below.

Scenario The user designs his security policies or, if they are not competent enough, picks the predefined one from the operator. The policy is formalized in the logical language 2D-LTL. Then the policies are automatically compiled in the form suitable for the runtime monitoring. The compiled policies are deployed at the device. When the application is downloaded and installed at the device it might be inlined with any of the user policies. After the user selects the policy the application is rewritten and the calls to the monitor are inserted before and after each call to the method mentioned in the policy. Then when the application is run the monitor checks whether its execution satisfies the policy, and if the violation is detected a security exception is raised in the application. If the application does not handle the security exceptions then it terminates. Otherwise, if the developers took the security exceptions into consideration the application may be able to proceed (or at least to terminate gracefully, notifying the user about the situation). But in any case the execution that violates the policy will not be possible.

6 Related Work and Conclusion

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

The limitation of the *Sandbox Security Model* [14] is that it can provide security but only at the cost of unduly restricting the functionality of mobile code (e.g., the code is not permitted to access any files).

Cryptographic code-signing is widely used for certifying the origin (i.e. the producer) of mobile code and its integrity. However, 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).

The *Proof Carrying Code (PCC)* approach [15] launched the idea that untrusted code is accompanied by additional information (a proof) that aids in verifying its safety. The traditional approach to PCC based on type theory is problematic for two main reasons noted by Sekar et al. [16]. A practical difficulty is that automatic proof generation for complex proper-

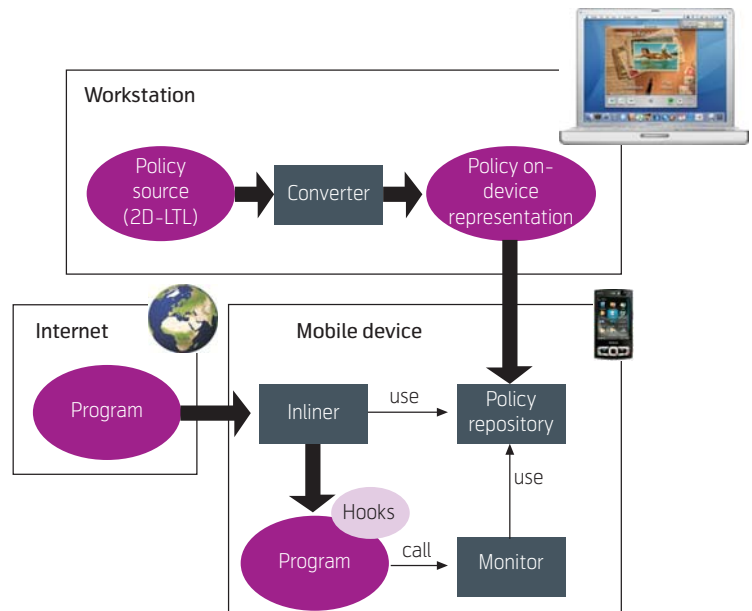


Figure 12 Usage Model

ties is still a daunting problem, requiring expertise in logic and theorem proving that is beyond the reach of normal developers. A more fundamental difficulty is that the type-theory approach is based on an unrealistic assumption [16]: the safety proof is based on a type system and this type system cannot be configured for each individual policy. This appears as an impractical assumption since security may vary considerably across different consumers and their operating environments.

Model Carrying Code (MCC) requires a producer to furnish a model regarding the safety of mobile code [12]. With MCC, this additional information captures the security-relevant behavior of the code. The major limitation is that MCC has not fully developed the issue of contract matching and has limited itself to finite state automata which are too simple to describe realistic policies.

In this article we have presented the Security-by-Contract (SxC) framework, which aims to build the basis for the opening of the software market of nomadic devices (from smartphones to PDA). The key intuition is the concept of *security contract*, a mobile contract that an application carries with itself. The rationale of the approach is that a digital signature should not just certify the origin of the code but rather bind together the code with a contract. We argue that Security-by-Contract would provide semantics for digital signatures on mobile code thus being a step in the transition from trusted code to trustworthy code. SxC will not replace, but enhance today's security mechanisms, and will provide a flexible, simple and scalable security mechanism for future mobile systems

References

- 1 LaMacchia, B, Lange, S. *.NET Framework security*. Addison Wesley, 2002.
- 2 Gong, L, Ellison, G. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.
- 3 Meyer, B. *Building bug-free O-O software: An introduction to Design by Contract(TM)*. Available at <http://archive.eiffel.com/doc/manuals/technology/contract/>.
- 4 Hilty, M, Pretschner, A, Schaefer, C, Walter, T. Usage control requirements in mobile and ubiquitous computing applications. **In: Proc. of the Int. Conf. on Sys. and Net. Comm. (ICSNC 2006)**, 27. IEEE Press, 2006.
- 5 MOBIUS Project Team. *Framework and application-specific security requirements*. Public Deliverable of EU Research Project D1.2, MOBIUS – Mobility, Ubiquity and Security. Report available at <http://mobius.inria.fr>, 2006.
- 6 Zobel, A, Simoni, C, Piazza, D, Nuez, X, Rodriguez, D. *Business case and security requirements*. Public Deliverable D5.1.1, EU Project S3MS. Report available at www.s3ms.org, October 2006.
- 7 Aktug, I, Naliuka, K. Conspec – a formal language for policy specification. **In: Proc. of the 1st Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007)**, 2007.
- 8 Massacci, F, Naliuka, K. Towards Practical Security Monitors of UML Policies for Mobile Applications. **In: Proceedings of ARES'08**, 1112-1119. IEEE Press, 2008.
- 9 Paul, N, Evans, D. .NET Security: Lessons Learned and Missed from Java. **In: Proc. of ACSAC'04**, 2004.
- 10 Bielova, N, Dalla Torre, M D, Dragoni, N, Siahaan, I. Matching Policies with Security Claims of Mobile Applications. **In: Proceedings of ARES'08**, 128-135. IEEE Press, 2008.
- 11 Massacci, F, Siahaan, I. *Matching midlet's security claims with a platform security policy using automata modulo theory*. NordSec, 2007.
- 12 Sekar, R, Venkatakrishnan, V N, Basu, S, Bhatkar, S, DuVarney, D C. Model-carrying code: a practical approach for safe execution of untrusted applications. **In: Proceedings of the 19th ACM symposium on Operating systems principles (SOSP-03)**, 15-28. ACM Press, 2003.
- 13 Clarke, E M, Grumberg, O, Peled, D A. *Model Checking*. The MIT Press, 2000.
- 14 Gong, L. Java Security: Present and Near Future. *IEEE Micro*, 17 (3), 14-19, 1997.
- 15 Necula, G C. Proof-Carrying Code. **In: Proc. of POPL '97**, 106-119, 1997. ACM Press.
- 16 Sekar, R, Ramakrishnan, C R, Ramakrishnan, I V, Smolka, S A. Model-Carrying Code (MCC): a New Paradigm for Mobile-Code Security. **In: NSPW '01: Proceedings of the 2001 Workshop on New security paradigms**, 23-30, New York, NY, USA, 2001. ACM Press.

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). His research interests are in service-oriented computing, multi-agent systems and computer security.

ndra@imm.dtu.dk

Fabio Massacci obtained his M.Eng. in 1993 and his Ph.D. in Computer Science and Engineering at University of Rome 'La Sapienza' in 1998. He visited Cambridge University in 1996-1997. He joined University of Siena as Assistant Professor in 1999 and was visiting researcher at IRIT Toulouse in 2000. In 2001 he joined the University of Trento as Associate Professor and since 2005 is full professor. He is visiting scientist at SINTEF in Norway. In 2001 he received the Intelligenza Artificiale award, a young researchers' career award from the Italian Association for Artificial Intelligence. He is a member of ACM, IEEE, and a chartered engineer. His research interests are in automated reasoning at the crossroads between requirements engineering and computer security. He has co-authored more than 70 papers in refereed journals and conferences. His h-index at 12/2007 is 18.

www.massacci.org / massacci@disi.unitn.it