A Security-by-Contracts Architecture for Pervasive Services

Fabio Massacci, Nicola Dragoni, and Ida S.R. Siahaan

DIT, Universitá di Trento, via Sommarive 14, 38050 Povo, Trento, Italy

1 Security-by-Contract (S×C)[3]

The paradigm of pervasive services [1] envisions a nomadic user traversing a variety of environments and seamlessly and constantly receiving services from other portables, handhelds, embedded or wearable computers. 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. These *pervasive client downloads* will appear because service providers will try to exploit the computational power of the nomadic devices to make a better use of the services available in the 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 nomadic host.

S×C **Framework.** The framework of S×C is shaped by four stake-holders: mobile operator, service provider or developer, mobile user and third party security service providers. Application developers are responsible to provide a *contract*, i.e. a formal, complete and correct specification of the behavior of an application for what concerns relevant security actions (Virtual Machine (VM) API Calls, Operating System Calls). Each "application" consists of four components: *executable code*, *run-time level contract*, *proof of compliance*, and *application credentials*. By signing the code the developer binds the code with the stated claims on its security-relevant behavior thus providing a semantics to digital signatures. An example of a contract is "After Personal Information Management (PIM) was opened no connections are allowed".

Users and mobile phone operators are interested in that any software deployed on their platform is secure by declaring security 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). An example of policy is "After PIM was accessed only secure connections can be opened i.e. url starts with "https://" ".

A contract should be negotiated and enforced during development, at time of delivery and loading, and during execution of the application code by the mobile platform. Fig. 1 shows the phases of the S×C life-cycle. S×C security architecture (Fig. 2) has two goals: supporting the application and service life cycle by guaranteeing the security of the channel between parties as well as authenticity of the parties and non-repudiation

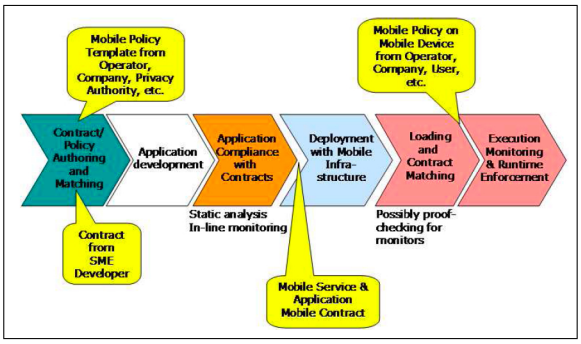

Fig. 1. Application/Service Life-Cycle

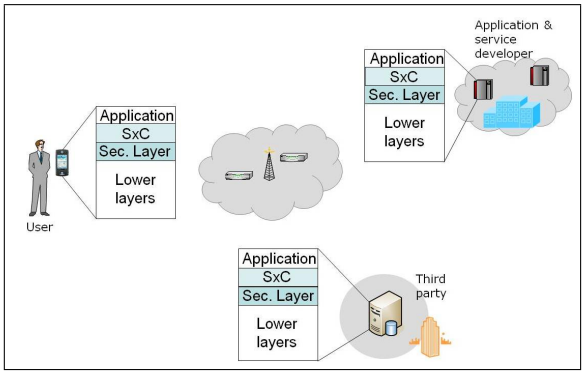

Fig. 2. S×C Architecture

of communication actions for charging and billing, and enabling trust relationships between stakeholders, i.e. authenticity and integrity of exchanged data elements.

2 Automata Modulo Theory ($\mathcal{AMT}$) [4]

We solved the problem of matching the security claims of the code with the security desires of the platform of S×C in [2] with only a meta-level algorithm showing how we can combine policies at different levels of details.The actual mathematical structure and algorithm to do the matching is specified in [4]. The key idea is based on the introduction of the concept of *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ enables us to define very expressive and customizable policies as a model for *security-by-contract* as in [2] and model-carrying code [6] by capturing the infinite transitions into finite transitions labeled as expressions in defined theories.

To represent a security behavior, provided by the contract and desired by the policy, a system can be represented as an automaton where transitions corresponds to the in-

voked methods as in the works on model-carrying code [6]. In this case, the operation of contract matching is a *language inclusion* problem.

$\mathcal{AMT}$ **Theory.** The theory of $\mathcal{AMT}$ is a combination of the theory of Büchi Automata (BA) with the Satisfiability Modulo Theories (SMT) problem. SMT problem pushes the envelope of formal verification based on effective SAT solvers. In contrast to classical security automata we prefer to use BA because besides safety properties, there are also some liveness properties which have to be verified. An example of liveness is "The application uses all the permissions it requests".

Definition 1 (Automaton Modulo Theory ($\mathcal{AMT}$)). *A tuple $A_{\mathcal{T}} = \langle E, S, q_0, \Delta_{\mathcal{T}}, F\rangle$ where E is a set of formulae in the language of the theory $\mathcal{T}$, S is a finite set of states, $q_0 \in S$ is the initial state, $\Delta_{\mathcal{T}} : S \times E \rightarrow 2^S$ is labeled transition function, and $F \subseteq S$ is a set of accepting states.*

$\mathcal{AMT}$ operations for intersection and complementation require that the theory is closed under intersection and complementation (union is similar to the standard one). We consider only the *complementation of deterministic* $\mathcal{AMT}$, because in our application domain all security policies are naturally deterministic (as the platform owner should have a clear idea on what to allow or disallow) (further details in [4]).

On-the-Fly State Model Checking with Decision Procedure. We are interested in finding counterexamples faster and we combine algorithm based on Nested DFS [5] with decision procedure for SMT. The algorithm takes as input the midlet's claim and the mobile platform's policy as $\mathcal{AMT}$ and then starts a DFS procedure over the initial state. When a suspect state which is an accepting state in $\mathcal{AMT}$ is reached we have two cases. First, when a suspect state contains an error state of complemented policy then we report a security policy violation without further ado. Second, when a suspect state which is an accepting state in $\mathcal{AMT}$ does not contain an error state of complemented policy we start a new DFS from the suspect state to determine whether it is in a cycle, in other words it is reachable from itself. If it is, then we report availability violation.

Theorem 1. *Let the theory $\mathcal{T}$ be decidable with an oracle for the SMT problem in the complexity class $\mathcal{C}$ then:*

1. *The non-emptiness problem for $\mathcal{AMTT}$ is decidable in $LIN - TIME^{\mathcal{C}}$.*
2. *The non-emptiness problem for $\mathcal{AMTT}$ is $NLOG - SPACE^{\mathcal{C}}$.*

References

1. J. Bacon. Toward pervasive computing. *IEEE Perv.*, 1(2):84, 2002.
2. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of EuroPKI*, 2007.
3. N. Dragoni, F. Massacci, C. Schaefer, T. Walter, and E. Vetillard. A security-by-contracts architecture for pervasive services. In *Proc. of SecPerU*, 2007.
4. F. Massacci and I. Siahaan. Matching Midlet's Security Claims with a Platform Security Policy using Automata Modulo Theory. *NordSec*, To Appear.
5. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. Tech Rep 2004/06, Univ. Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, 2004.
6. R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proc. of SOSP*, 2003.