View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by PUblication MAnagement

Extending Security-by-Contract with Quantitative Trust on Mobile Devices

Gabriele Costa
Dipartimento di Informatica
Università di Pisa
costa@di.unipi.it

Nicola Dragoni
DTU Informatics
Technical University of Denmark
Email: ndra@imm.dtu.dk

Aliaksandr Lazouski
Dipartimento di Informatica
Università di Pisa
lazouski@di.unipi.it

Fabio Martinelli
IIT-CNR
Pisa Research Area
fabio.martinelli@iit.cnr.it

Fabio Massacci
Dipartimento di Ingegneria e Scienza dell'Informazione
Università di Trento
massacci@disi.unitn.it

Ilaria Matteucci
IIT-CNR
Pisa Research Area
ilaria.matteucci@iit.cnr.it

inria-00536705, version 1 - 16 Nov 2010

***Abstract*—Security-by-Contract (S×C) is a novel paradigm providing security assurances for mobile applications. In this work, we present an extension of S×C enriched with an automatic trust management infrastructure. Indeed, we enhance the already existing architecture by adding new modules and configurations for contracts managing. At deploy-time, our system decides the run-time configuration depending on the credentials of the contract provider. Roughly, the run-time environment can both enforce a security policy and monitor the declared contract. According to the actual behaviour of the running program our architecture updates the trust level associated with the contract provider. The main advantage of this method is an automatic management of the level of trust of software and contract releasers.**

I. Introduction

Mobile Java applications (MIDlets) offer a clear example of fixed trust relationship. As a matter of fact, a MIDlet is a software released by some vendor that clients download and install on their device. The serious constraints on the resources of mobile devices (*e.g.*, CPU, memory, battery) make several security mechanisms practically infeasible. The current technique for providing security assurances to mobile device users is based on software certification. Roughly, *certification authorities* (CAs) provide software developers/releasers with signed certificates. Software companies acquire certificates having a temporary validity and use them for signing their applications. When installing a signed MIDlet, the user's system checks if the attached certificate is valid and which CA released it. If the certificate source is trusted, the application can run and access local resources with no restrictions, otherwise the user is alerted about the potential danger deriving from installing the MIDlet and required for providing explicit permission to each single access operation.

The set of trusted CAs can be statically defined, *i.e.*, the device manufacturer specifies a unmodifiable list of accepted certificates sources, or user-modifiable, *i.e.*, users can add and remove CAs from the list. The certificate-based approach has several, well known drawbacks. Mainly, it implements a white list strategy. While certified MIDlets have all the privileges they need, uncertified applications have very little access to the system independently from their actual behaviour, leading to a significant reduction of their usability. On the other hand, executing a malicious, signed application can have obvious, dramatic consequences. There are many ways in which this attack can take place. A simple attacking scenario is based on the user's unawareness about security. Indeed, a device owner wanting to install a MIDlet could decide to insert its CA among the trusted ones. This scenario is becoming popular, for instance, with local providers offering small, contextual applications (*e.g.*, catalogues, interactive guides). Often, MIDlet spots dispatch unsigned or self-certified applications to users moving inside some area of interest (*e.g.*, a museum).

Another danger arises from the hierarchical structure of certificates. In fact, when purchasing a certificate, the owner is often authorized to produce and distribute sub-certificates. The features of a sub-certificate depend on the structure of the original one (*e.g.*, a certificate can generate sub-certificates with an expiration date lower or equal to its own). For instance, an attacker acquiring a certificate can use it for signing a malicious MIDlet. Then, after detecting the attack, it should be possible, analysing the certificate, to trace back the certificate history and discover what went wrong in the sub-certificates chain. However, this is a reactive approach that can lead to identifying misbehaving entities (CAs, developers, vendors), while, in general, a proactive solution would be preferable.

For these reasons we advocate a contract-based approach for mobile applications trust management. In our model contracts are in charge of providing guarantees on the correct behaviour of programs. Contracts are automatically produced by any provider and attached to the code. The counterpart of contracts are security policies. Policies define which behaviours are considered to be safe. To implement

Work partially supported by EU project FP7-231167 "CONNECT".

inria-00536705, version 1 - 16 Nov 2010

this strategy, we present a contract monitoring framework responsible for verifying whether a running application respects its contract. When an attack, namely an attempt to violate a contract, is detected, our system reacts immediately by enforcing a security policy and preventing the attack from being actually performed. Moreover, a contract breaking causes an automatic modification of the trust relationship between the device and the authority providing the contract. In this way, our system can immediately react to threats and prevent further attacks coming for the same source. Furthermore, the proposed framework offers a high degree of flexibility providing applications clients with a reliability feedback and assuring security guarantees also under pervasive, contextual mobility conditions.

This paper is structured as follows: Section II recalls some background notions. Section III presents our proposed extension of the security by contract paradigm with trust measurement. Section IV relates our contribution to previous ones already in literature and Section V provides the conclusion of the paper and our future work.

II. Background

In this section, we recall some notions about the *Security-by-Contract* paradigm [1].

A. Security-by-Contract paradigm

The *Security-by-Contract* (S×C) [1] paradigm provides a full characterisation of the contract-based interaction. Roughly, the code released by a provider is annotated with a contract. When a client receives an application verifies whether the code and the contract actually match by an *evidence checking* procedure. If the check fails then the user can decide to delete the MIDlet or to enforce a security policy on it by exploiting the *monitoring* infrastructure. Otherwise, the system can proceed to verify whether the contract (correctly representing the application) satisfies the user's policy. Once again, if this step fails the solution consists in enforcing the active policy on the execution. Finally, if the previous checks were positively passed, the MIDlet can be executed with no active runtime monitor.

The Contract-Policy matching function ensures that any security relevant behaviour allowed by the contract is also allowed by the policy. This matching could be done w.r.t. different behavioural relation, *e.g.*, language inclusion [2] or simulation relation [3]. This matching function allows the user that is going to execute the MIDlet to understand if the behaviour of the applet itself is compliant with the set of policies he has on his device or not without running it.

The enforcing approach has been shown to be feasible on mobile devices. In particular two techniques have been detailed in the literature and exploited for experiments and tools: JVM customization [4] and bytecode in-lining [5]. Briefly, the first replace the standard JVM with a modified one dispatching signals to the monitoring agent whenever a program makes a call to (a subset of) the system APIs. The second instruments the sequence of bytecode instructions with invocations to the security policy monitor making the program send security signals at run-time. Both approaches use an external component, namely a *Policy Decision Point* (PDP), holding the set of rules that compose the security policy. Moreover the PDP reads the current device state (battery consumption, link strength, available credit) through dedicated internal components. When the PDP receives a request for an action violating the security policy, it answers denying the necessary permission. Then, the system reacts by throwing an exception.

B. Trust management

Trust management techniques are used in systems where some level of uncertainty exists upon components behaviour. In case of mobile platforms, a user downloading an unknown application can not demonstrate a knowledge of its behaviour prior to execution. Hence, trustworthy applications guarantee a correct behaviour with respect to user's security policy and can be authorized to access a mobile device. Schneider et al. [6] outlined the *axiomatic*, *analytic*, and *synthetic* basis to measure application's trust value.

Axiomatic trust is based on security assertions done by trusted principals about the application. This is a classical scenario for mobile platforms where applications are accompanied with certificates signed by some principals. The comprehensive model of axiomatic trust management for mobile devices was presented in [7]. The model implemented a role-based trust management framework [8] and assumed a delegation of authority to issue security assertions. Each security assertion was supplemented with a weight expressing quantitative trust put by the issuer on the assertion. For instance, the assertion, encoded as $A.f(v) \leftarrow D$, states that principal A trusts D for performing functionality f with degree v, where $v \in [0;1]$. Principals could form more complex assertions using RT syntax, for example, containment and delegation assertions, etc. Due to transitive and distributed model of authority, the approach allows to deduce principal's trustworthiness without having direct relations. For example, a user can reason to execute an application having three assertions. The first states that a principal A guarantees application's trustworthiness $A.trusted(v_A) \leftarrow MIDlet_x$; the second constitutes that a principal B knows A, $B.trusted(v_B) \leftarrow A$; and, finally, the user trusts the principal B, $User.trusted(v_U) \leftarrow B$. The resulting trust weight assigned by the user to the application is a multiplication of trust weights in the delegation chain, $User.trusted(v_A \cdot v_B \cdot v_U) \leftarrow MIDlet_x$. The approach in [7] provided an algorithm to calculate trust weights but nothing was said on the adjustment of weights in assertions.

Obviously, the axiomatic trust has pros and cons. The main drawback is that it deals with assertions about the history of previous interactions and does not guarantee a

inria-00536705, version 1 - 16 Nov 2010

trustworthy behaviour of the principal after granting the access. *Analytic trust* persuades to trust certain behaviours of a principal in the future using some analysis. We consider a security-by-contract as the best example of the analytic trust. A contract contains a specification of application's behaviour which can be easily verified by the user before the real execution.

Syntactic trust addresses a run-time monitoring of unknown applications. Thus, the application execution is allowed but constrained by a security policy. Accesses violating the security policy are forbidden and the monitor terminates application's execution.

To the best of our knowledge, a union of these aspects of trust was never addressed in the previous work. The traditional approach concerns on axiomatic trust only, while the S3MS project introduces analytic and syntactic trust models. Instead, we employ all these aspects of trust to reason on application's trustworthiness, and we are also interested in how these aspects of trust co-exist and influence on each other in the context of mobile platforms.

III. Our Goal

The main novelty of our approach consists in integrating the S×C paradigm with a monitoring infrastructure for trust management by exploiting Role-based Trust Management Language (RTML) to deal with both trust and reputation management. In particular we aim at automatizing the updating of the trust relationship between users and application/contract providers. As a matter of fact, one crucial point of the S×C architecture is the checking of the relation that exists between the applet and its contract. Nowadays the mobile code is run if its origin is trusted. This means that we can only reject or accept the signature of the application/contract provider.

Here we propose an extension of the existing architecture by adding a component for the contract monitoring that allows us to check if the execution of the application adheres to the contract of the application itself and, according to the answer, we update the level of trust of the provider.

Our strategy takes place in two phases: at deploy-time by setting the monitoring state and at run-time by applying the contract monitoring procedure for adjusting the provider trust level.

A. Deployment Architecture

The S×C paradigm works according the Application/Service Life-Cycle depicted in Fig. 1. The users have a device on which they apply their own security definition by designing personal policies or retrieving them from some provider, *e.g.*, the device manufacturer. When he downloads an application the system automatically checks the formal correspondence between code and contract (Check Evidence). This step is intended to provide a formal proof that the contract effectively denotes every possible behaviour

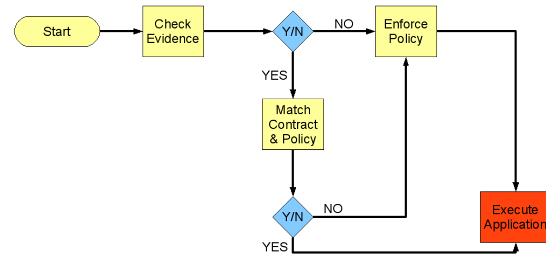


Figure 1: The Security-by-Contract Application lifecycle [9].

of the running program. This step can be implemented, for instance, using the *model-carrying code* [10] method.

If the result is negative then the monitor runs to enforce the policy (Enforce Policies), otherwise a matching between the contract and the policy is performed to establish if the contract is compliant with the policy. If it is the case than the application is executed (Execute Application), otherwise the policy is enforced again (Enforce Policies).

This security model does not require the software provider to be a trusted entity and simply relies on the correctness of local, internal components (*i.e.*, Check Evidence and Contract-Policy Matching). Here, we deploy this model with

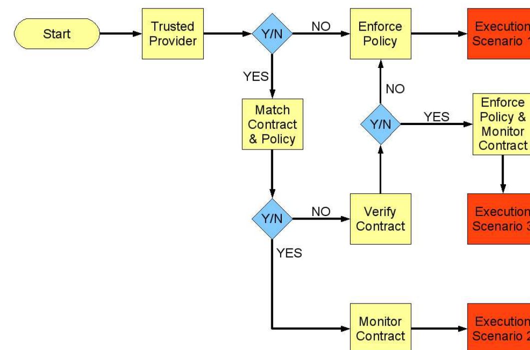


Figure 2: The Extended Security-by-Contract Application life-cycle.

a framework with quantitative trust in such a way that it is possible to update the level of trust dynamically according to the adherence between the real execution of the application and its contract. As matter of fact we extend the existing architecture by adding a *contract monitoring* that checks the compliance between the application and its contract. Hence the extended Application/Service Life-Cycle results as in Fig. 2. In fact we replace the component Check Evidence by a Trusted Provider that decides according to the level of

inria-00536705, version 1 - 16 Nov 2010

trust of the provider. If the provider is untrusted then the policy is enforced, otherwise the contract-policy matching is performed.

Two new scenarios arise:

1) The contract satisfies the policy. In this case, even if it is not useful for policy enforcement, we monitor the contract on a statistical basis. If the contract is satisfied we report 'OK' and the level of trust remains unchanged. Otherwise, if the contract is violated we report a trust violation and we continue to monitor the policy (the policy starts at the end of the recorded sequence of states monitored).
2) The contract does not satisfy the policy. Firstly, we check if the application adheres to the contract. If it is not the case, we directly enforce the policy, otherwise both the contract monitoring and the policy enforcement are performed at the same time.

Let us notice that, in the second scenario, the verification component plays a central role. Indeed, a negative result denotes that a trusted provider released a fake contract. Clearly, this event means that the previous trust value must be updated. Moreover, this component could be missing or unavailable (*e.g.*, due to the limited device resources). In this case we reduce ourselves to the mixed (*i.e.*, monitoring and enforcing) scenario.

B. Trust adjustment via contract monitoring

We outline how trust measures assigned to security assertions can be adjusted as a result of a contract monitoring strategy. Indeed, trust measures associated with the provider concern on the contract goodness mainly. Updated trust measures will influence on future interactions with an application and contract providers. In other words, our system penalizes the provider more when the contract does not specify application's behaviour correctly, rather when the application itself contradicts user's security policy.

Here we present a possible extension of the monitoring infrastructure model proposed in [5] by making the *policy decision point* (PDP) also responsible for the contract monitoring operations and for the trust vector updating. Roughly, in [5] a monitoring infrastructure consists in a PDP that holds the actual security state and is responsible for accepting or refusing new actions and *Policy enforcement points* (PEPs) that are both in charge of intercepting actions to be dispatched to the PDP and preventing the execution of not allowed operations.

Starting from this model, we extend it by making the PDP also responsible for the contract monitoring operations and for the trust vector updating. According to [4], [5], we assume that both contracts and policies are specified through the same formalism. Hence, the policy enforcement configuration of the PDP keeps unchanged. The PDP must load application contracts as well as security policies dynamically. Moreover, it must be able to run under three different execution scenarios (Fig. 2): policy enforcement enabled, contract monitoring enabled or both.

The base enforcement scenario (execution scenario 1) is actually unchanged w.r.t. the standard usage of the classical PDP. Hence, no contract monitoring nor trust management operations are involved.

Main interest resides in the other two scenarios. The contract monitoring scenario applies to programs carrying a contract released by a trusted authority (see Section III-A). Similarly to the policy enforcement strategy, PEPs send event signals to the PDP. The main difference is that the PDP keeps in memory the program events trace. When a signal arrives, the PDP checks whether it is consistent with the monitored contract. If the contract is respected then the PDP updates its internal monitoring state and answers allowing the operation. Otherwise, if a violation attempt happens, the PDP reacts changing its state.

The first consequence of a contract violation is a decreasing of the trust weights of both, direct and transitive security assertions. Indeed, the contract monitor detected a fake execution of a trusted application w.r.t. its declared contract.

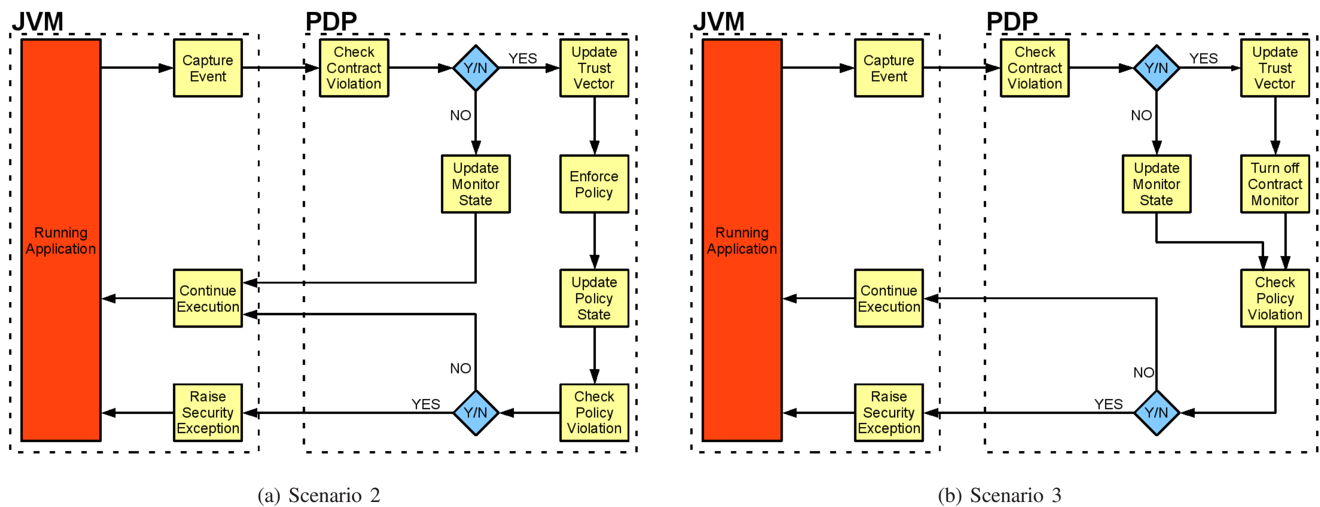
Secondly, the PDP changes from contract monitoring to policy enforcement configuration. Since an instance of the policy is always present, this operation does not imply a serious computational overhead. Afterwards, the policy state is updated using the execution trace recorded during the monitoring phase. This step, that can be time consuming, is necessary for verifying whether, breaking the contract, the application has also violated the policy. However, this computational cost, being the consequence of an extraordinary event, must be paid at most once. Indeed, when the PDP is performing both contract monitoring and policy enforcement, the current policy state is known. Finally, the execution continues with the PDP enforcing the policy starting from the last action, that is the event breaking the contract. Figure 3 shows the behaviour of the PDP performing the contract monitoring task in the two previously discussed scenarios.

Summing up, both execution scenario 2 and 3 check contract violations through the contract monitoring strategy described above and update providers' trust level. Such updates will influence future interactions with applications and contract providers. Trust measures associated with the provider concern on the contract goodness mainly. In other words, our system penalizes the provider more when the contract does not specify application's behaviour correctly, rather when the application itself contradicts user's security policy. Quantitative representation of this adjustment we consider to present in the future work.

IV. Related work

To the best of our knowledge, there is not much work about the integration of trust management and policy en-

inria-00536705, version 1 - 16 Nov 2010



(a) Scenario 2 (b) Scenario 3

Figure 3: The contract monitoring configurations

forcement for mobile code in literature. However, works about the integration of trust module into policy enforcement exist. In particular some work has been done for integrating trust management and fine grained access control in Grid Architecture. An attempt can be found in [11] where it is proposed an access control system the enhances the Globus toolkit with a number of features. This copes with the fact that access control policies and access rights management becomes one of the main bottleneck using Globus for sharing resource in a Grid architecture. Along this line of research [12] presents an integrated architecture, extending the previous one, with an inference engine managing reputation and trust credentials. This framework is extended again in [13] where it is introduced a mechanism for trust negotiating credential to overcome scalability problem. In this way the framework provided preserves privacy credentials and security policy of both users and providers. Even if the application scenario and the implementation are different, the basic idea consists in considering the trust as a metrics for deciding the reliability of an application provider.

Also [14] presents a reputation mechanism to facilitate the trustworthiness evaluation of entities in ubiquitous computing environments. It is based on probability theory and supports reputation evolution and propagation. The proposed reputation mechanism is also implemented as part of a QoS-aware Web service discovery middleware and evaluated regarding its overhead on service discovery latency. On the contrary, our approach is not probabilistic. We provide a method according to which we update the level of trust of an application provider.

Four main approaches to mobile code security can be broadly identified in the literature.

The sandbox model [15] limits the instructions available for use, with the weakness that it provides 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 [16], where permissions available for programs from a code source are specified through a security policy.

Cryptographic code-signing is used for certifying the origin (i.e. the producer) of mobile code and its integrity, typically by means of private/public keys to sign/verify the executable content. Several limitations of this approach can be identified [17]. A key weakness is that in the signing process it is not checked at all 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 (F-Secure Weblog (www.f-secure.com/weblog) 11/05/2007, "Just because it's Signed doesn't mean it isn't spying on you").

The *Proof-Carrying Code* (PCC) approach [18] 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, that the proof is valid and hence the foreign code is safe to execute. The PCC approach is problematic for several reasons [19], such as that automatic proof generation for complex properties is still a daunting problem, making the PCC approach not suitable for real mobile applications.

The *Model-Carrying Code* (MCC) 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 [10]. With MCC, this additional information takes the form of a model that captures the *security-relevant behaviour* of code, rather than a proof.

V. Conclusion and Future Work

In this paper we presented a contract-based trust management framework for mobile applications. At deploy-time,

inria-00536705, version 1 - 16 Nov 2010

the monitoring structure is decided depending on both the application contract and the credentials (*i.e.*, trust measure) of the contract releaser. The main novelty of our model consists in the contract monitoring scenario. At run-time, a trusted program violating its contract leads to a correction of the trust relationship with the provider and activates the policy enforcement configuration of our system.

Many future directions are viable. Mainly our trust management strategy is still a work in progress. Currently, trust weights can only decrease monotonically as a consequence of contract violations with the only exception of a direct intervention of the user. Also the contracting infrastructure can be further improved. As a matter of fact, in this work we only referred to single-contract applications. However, we can extend our model in order to accept more contract instances for a single program. This scenario seems to be realistic and open new directions of investigation.

Finally, similarly to [5], we plan to implement a working prototype for testing the practical feasibility of our approach.

Acknowledgement

We thank the anonymous referees of IMIS10 for valuable comments that helped us to improve this paper.

References

[1] N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, and E. Vetillard, "Security-by-contract (SxC) for software and services of mobile systems," in *At your service - Service-Oriented Computing from an EU Perspective.* MIT Press, 2008.

[2] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe, "Security-by-contract on the .net platform," vol. 13, no. 1. Oxford, UK, UK: Elsevier Advanced Technology Publications, 2008, pp. 25–32.

[3] P. Greci, F. Martinelli, and I. Matteucci, "A framework for contract-policy matching based on symbolic simulations for securing mobile device application," in *ISoLA*, 2008, pp. 221–236.

[4] A. Castrucci, F. Martinelli, P. Mori, and F. Roperti, "Enhancing java me security support with resource usage monitoring," in *ICICS*, 2008, pp. 256–266.

[5] G. Costa, F. Martinelli, P. Mori, C. Schaefer, and T. Walter, "Runtime monitoring for next generation java me platform," *Computers & Security*, July 2009.

[6] F. B. Schneider, K. Walsh, and E. G. Sirer, "Nexus authorization logic (nal): Design rationale and applications," Cornell Computing and Information Science Technical Report, Tech. Rep., 2009.

[7] D. Fais, M. Colombo, and A. Lazouski, "An implementation of role-base trust management extended with weights on mobile devices," *Electronic Notes in Theoretical Computer Science*, vol. 244, pp. 53 – 65, 2009, proceedings of the 4th International Workshop on Security and Trust Management (STM 2008).

[8] N. Li, J. C. Mitchell, and W. H. Winsborough, "Design of a role-based trust management framework," in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2002, pp. 114–130.

[9] N. Dragoni, F. Massacci, and K. Naliuka, "Security-by-contract(SxC) for mobile systems- or how to download software on your mobile without regretting it." Position Papers for W3C Workshop on Security for Access to Device APIs from the Web, December 2008.

[10] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney, "Model-carrying code: a practical approach for safe execution of untrusted applications," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 15–28.

[11] H. Koshutanski, F. Martinelli, P. Mori, L. Borz, and A. Vaccarelli, "A fine grained and x.509 based access control system for globus," in *OTM*. Springer, 2006, pp. 1336–1350.

[12] M. Colombo, F. Martinelli, P. Mori, M. Petrocchi, and A. Vaccarelli, "Fine grained access control with trust and reputation management for globus," in *OTM Conferences (2)*, 2007, pp. 1505–1515.

[13] H. Koshutanski, A. Lazouski, F. Martinelli, and P. Mori, "Enhancing grid security by fine-grained behavioral control and negotiation-based authorization," *Int. J. Inf. Sec.*, vol. 8, no. 4, pp. 291–314, 2009.

[14] J. Liu and V. Issarny, "An incentive compatible reputation mechanism for ubiquitous computing environments," *Int. J. Inf. Secur.*, vol. 6, no. 5, pp. 297–311, 2007.

[15] L. Gong, "Java Security: Present and Near Future," *IEEE Micro*, vol. 17, no. 3, pp. 14–19, 1997.

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

[17] N. Dragoni, F. Massacci, T. Walter, and C. Schaefer, "What the heck is this application doing? - a security-by-contract architecture for pervasive services," to appear in *Computers & Security*, Elsevier.

[18] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, Jan. 1997, pp. 106–119.

[19] 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*. New York, NY, USA: ACM Press, 2001, pp. 23–30.